Data Structure

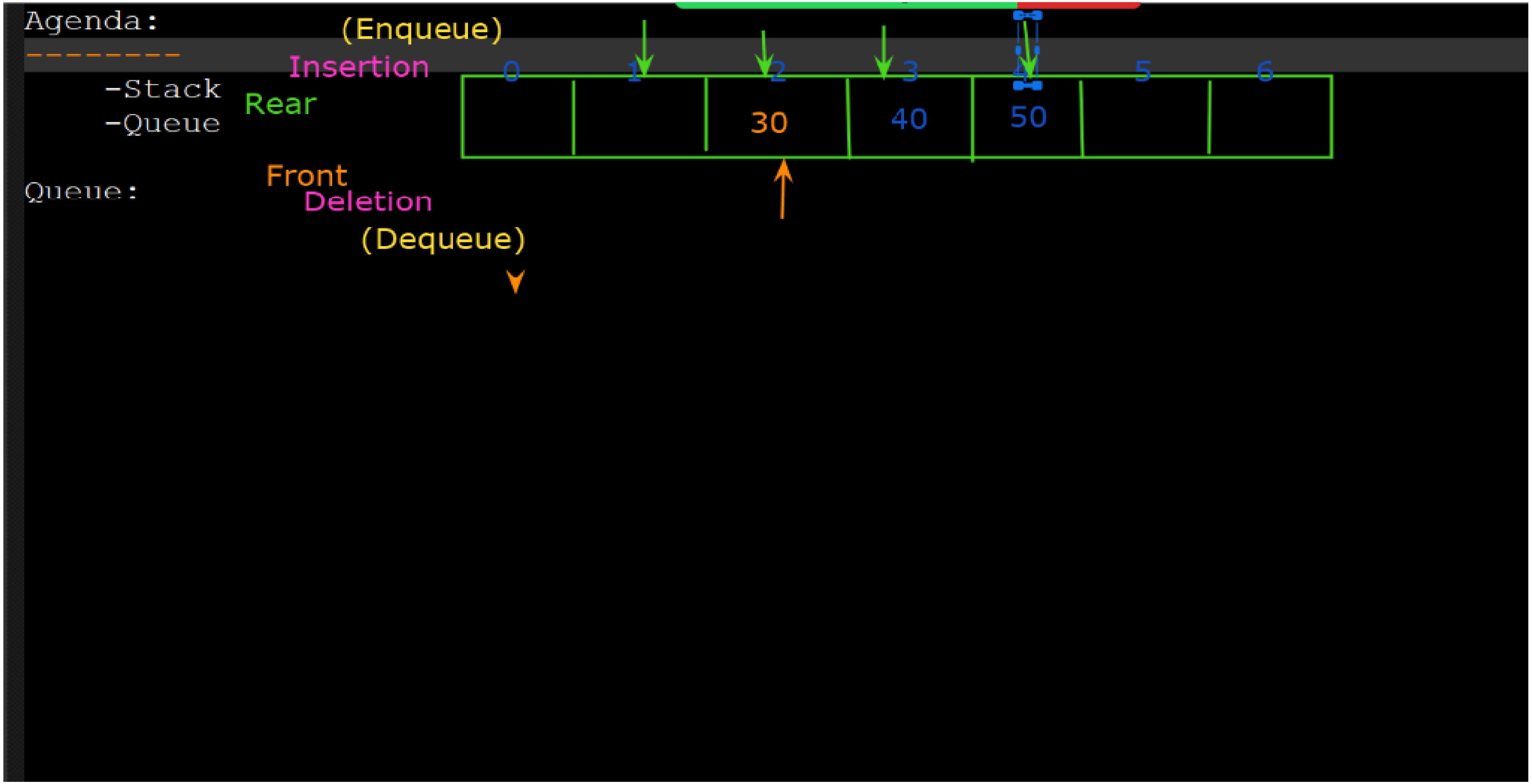
Sep23 : Day 9

Kiran Waghmare

CDAC Mumbai

Queue

Kiran Waghmare



Agenda:

-Stack

-Queue

Queue:

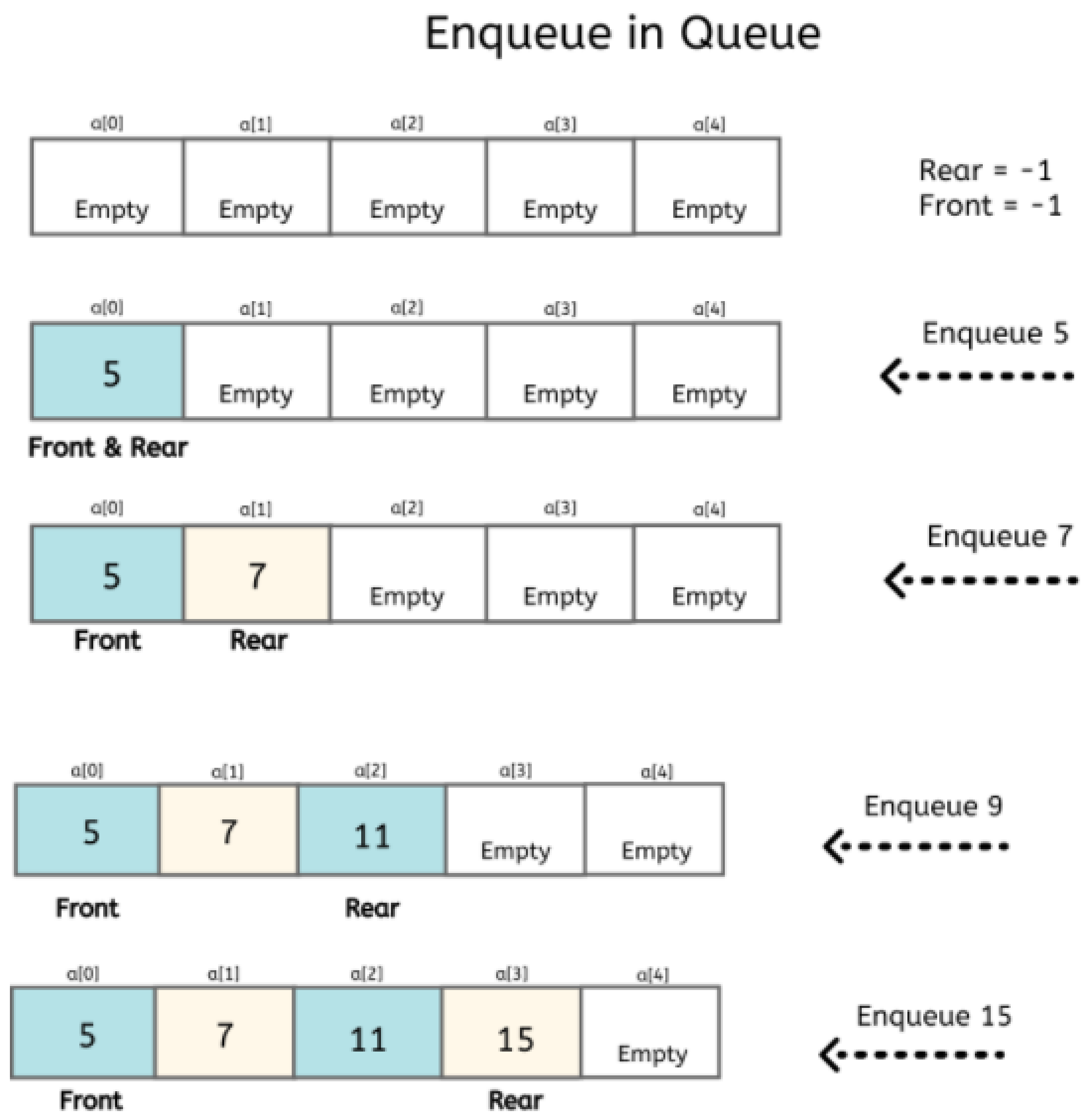
Queue

- Ordered collection of homogeneous elements.
- Non-primitive linear data structure.
- A new element is added at one end called rear end and the existing elements are deleted from the other end called front end.
- This mechanism is called First-In-First-Out (FIFO)
- Total no. of elements in queue = rear-front+1

1. Enqueue()

When we requirea to add an element to the Queue we perform Enqueue() operation.

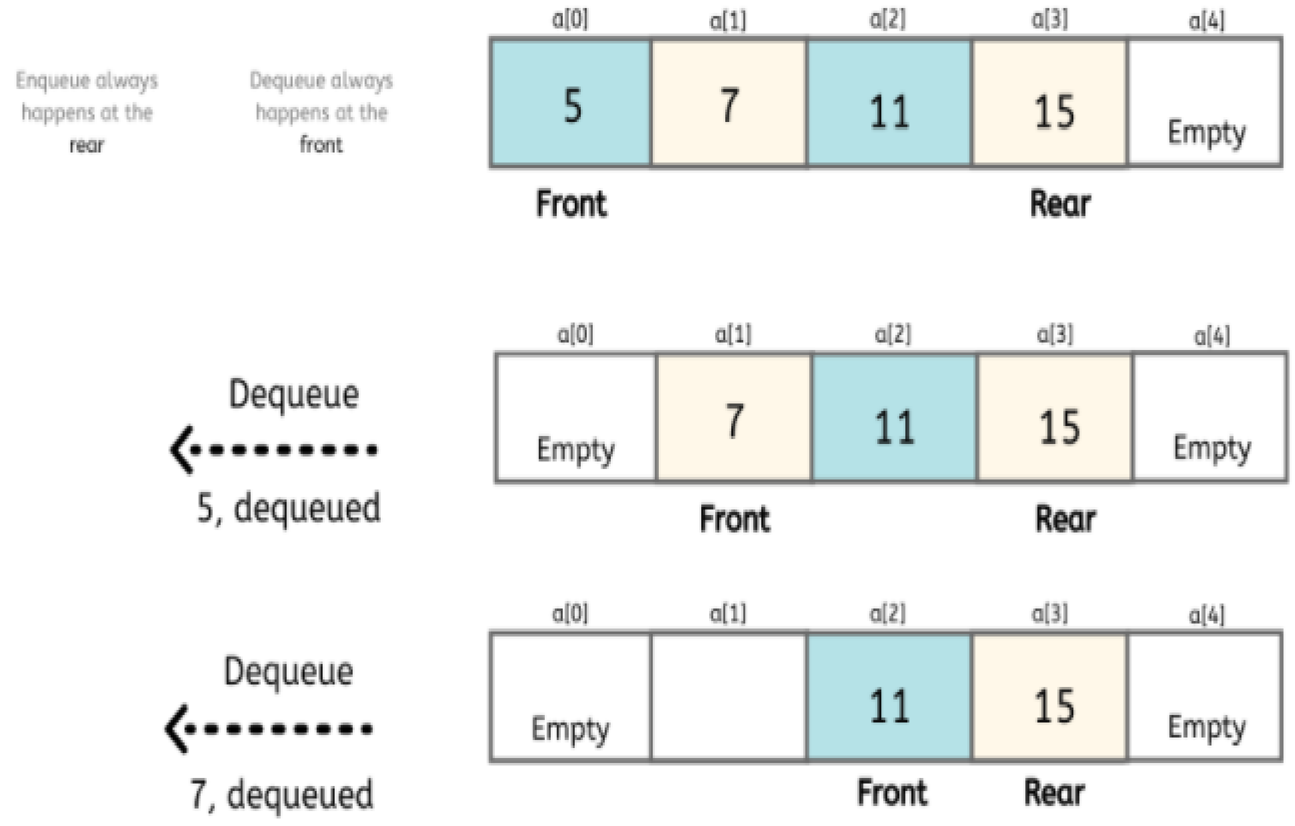
Push() operation is synonymous of insertion/addition in a data structure.



2. Dequeue()

When we require to delete/remove an element to the Queue we perform Dequeue() operation.

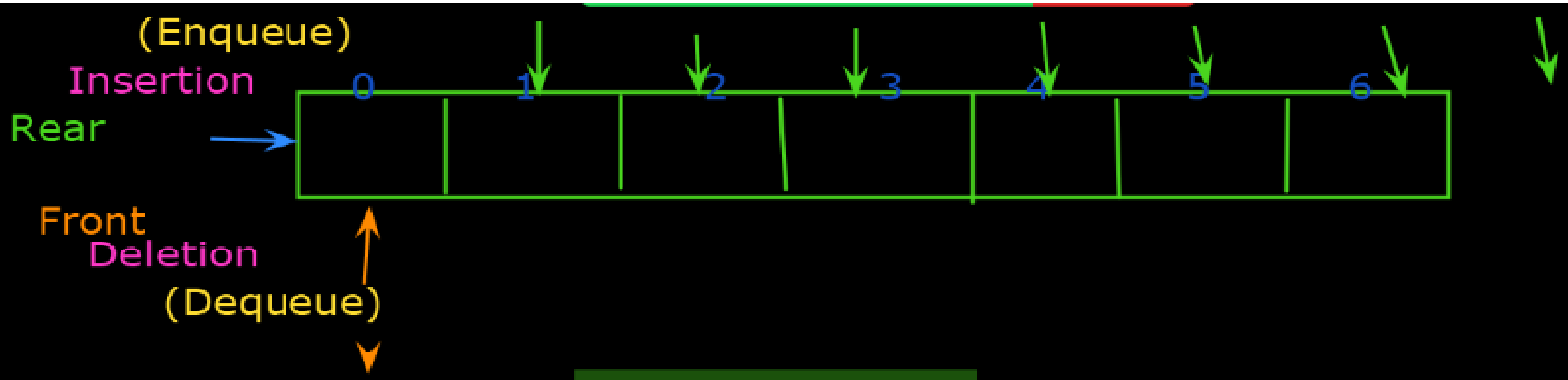
Dequeue() operation is synonymous of deletion/removal in a data structure.



Agenda:

- -Stack
- -Queue

Queue:

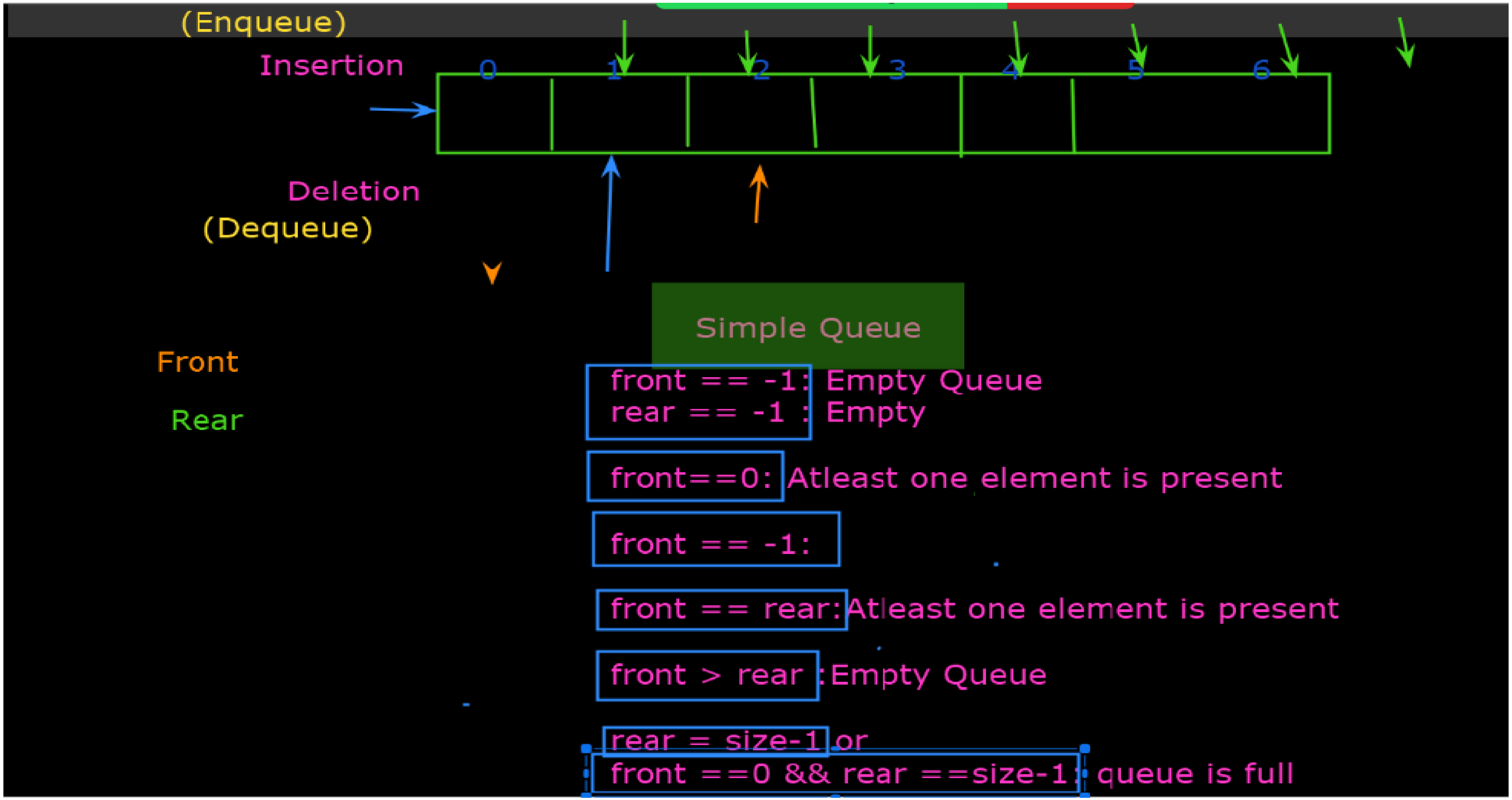


Simple Queue

-data structure
-Queue follows the rule : FIFO (First In First Out)
-Insertion operation is called as Enqueue
-Deletion operation is called as Dequeue
-Useing 2 pointers:
- -Rear: Inserting an element in queue
- -Front: Deleting an element in queue
- -Rear = -1
- -Front = -1/ 0

Operations on th Queue:

-Enqueue
-Dequeue



front == -1: Empty Queue
rear == -1 : Empty

front==0: Atleast one element is present

front == -1:

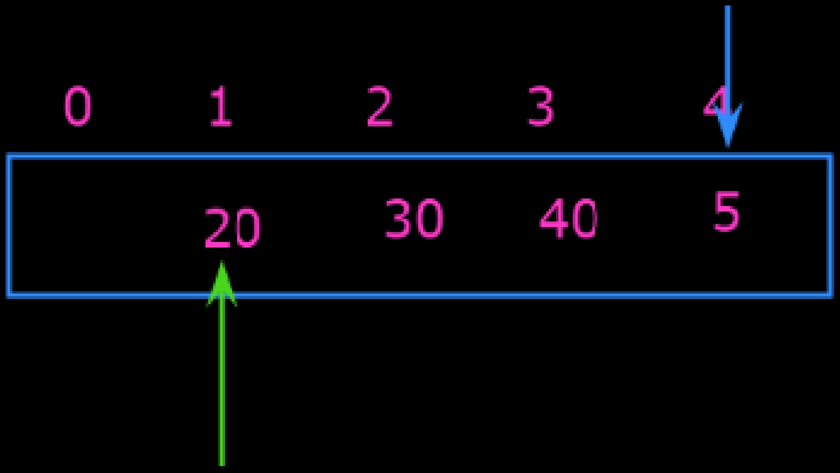
front == rear: Atleast one element is present

front > rear :Empty Queue

rear = size-1 or
front ==0 && rear ==size-1: queue is full

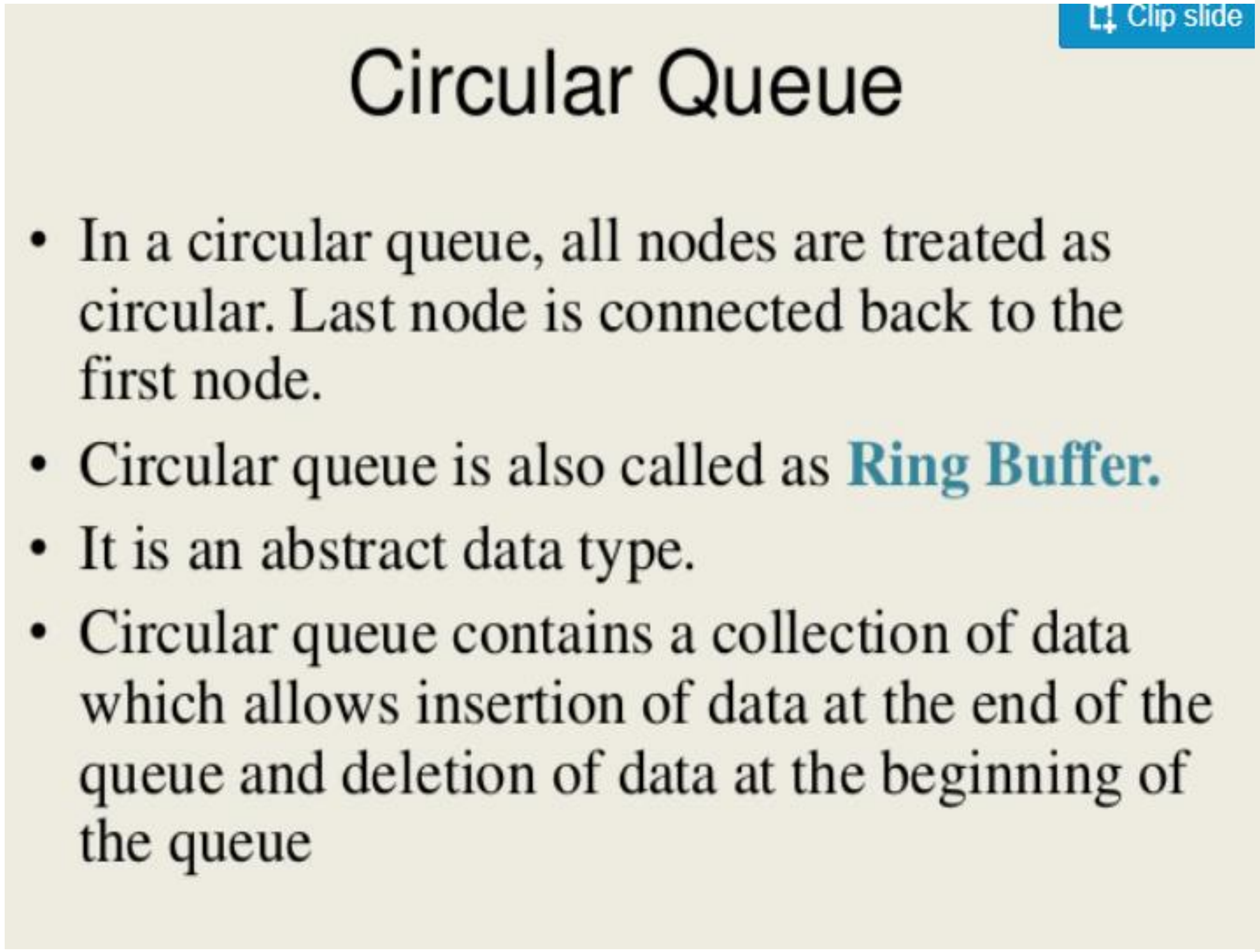
```
    {
        for(int i=front;i<=rear;i++)
        {
            System.out.println(Q[i]+" ");
        }
    }
    //print the values of rear & front

    public static void main(String args[])
    {
    }
}
```



Time complexity:

Enqueue: O(1)
Dequeue: O(1)



Circular Queue

- In a circular queue, all nodes are treated as circular. Last node is connected back to the first node.
- Circular queue is also called as **Ring Buffer.**
- It is an abstract data type.
- Circular queue contains a collection of data which allows insertion of data at the end of the queue and deletion of data at the beginning of the queue

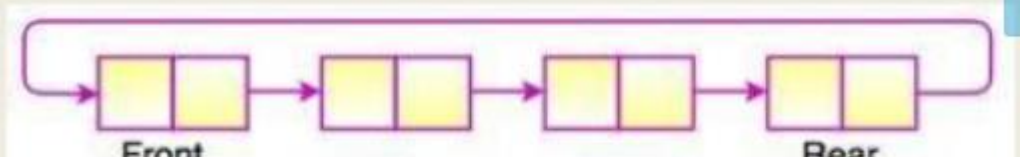


Fig. Circular Queue

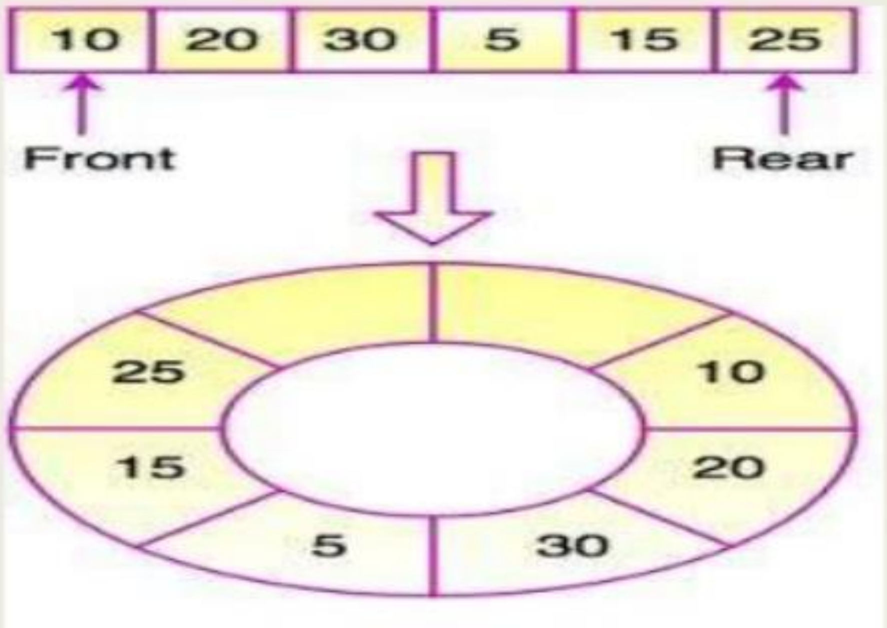
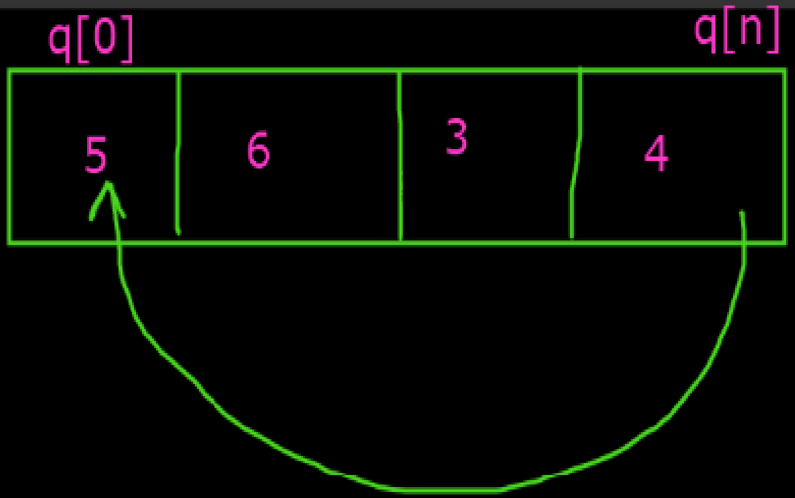


Fig. Circular Queue

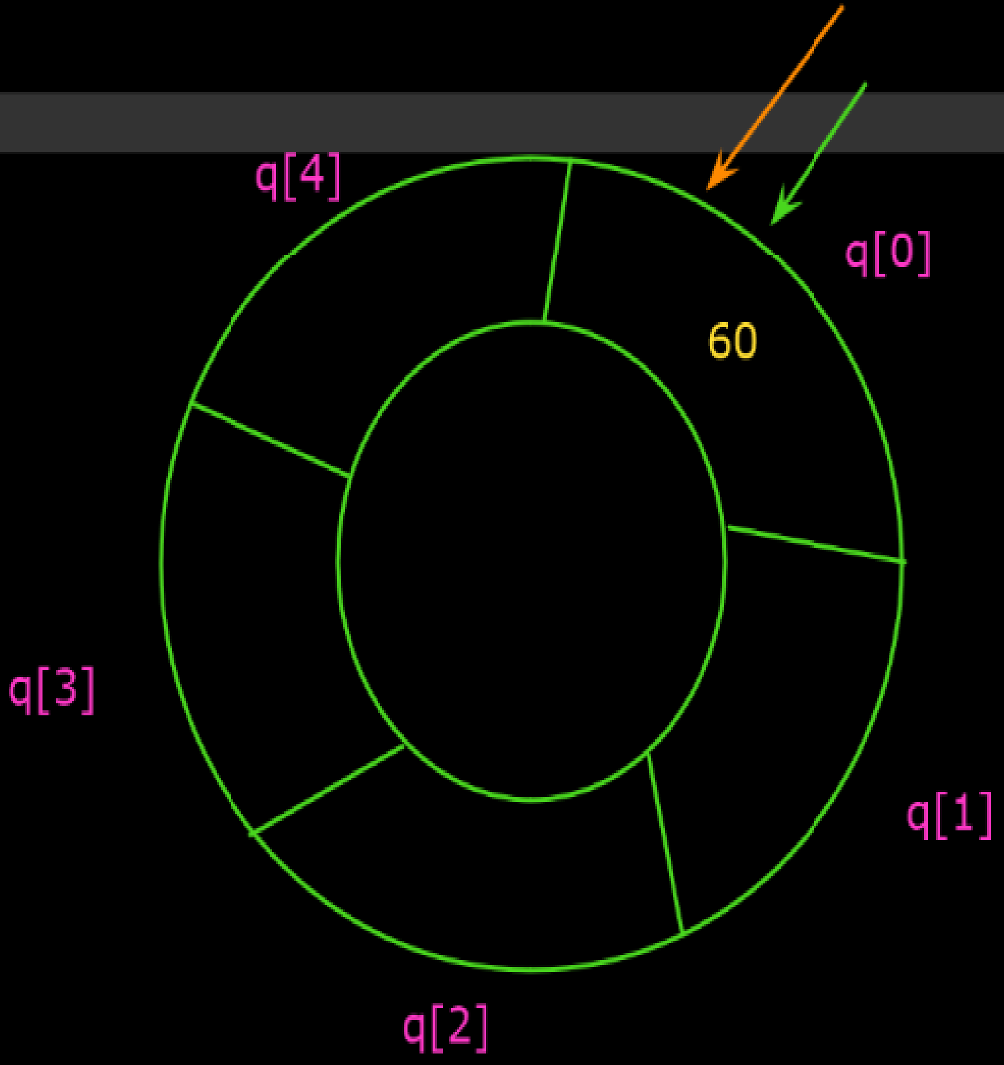
Circular Queue:

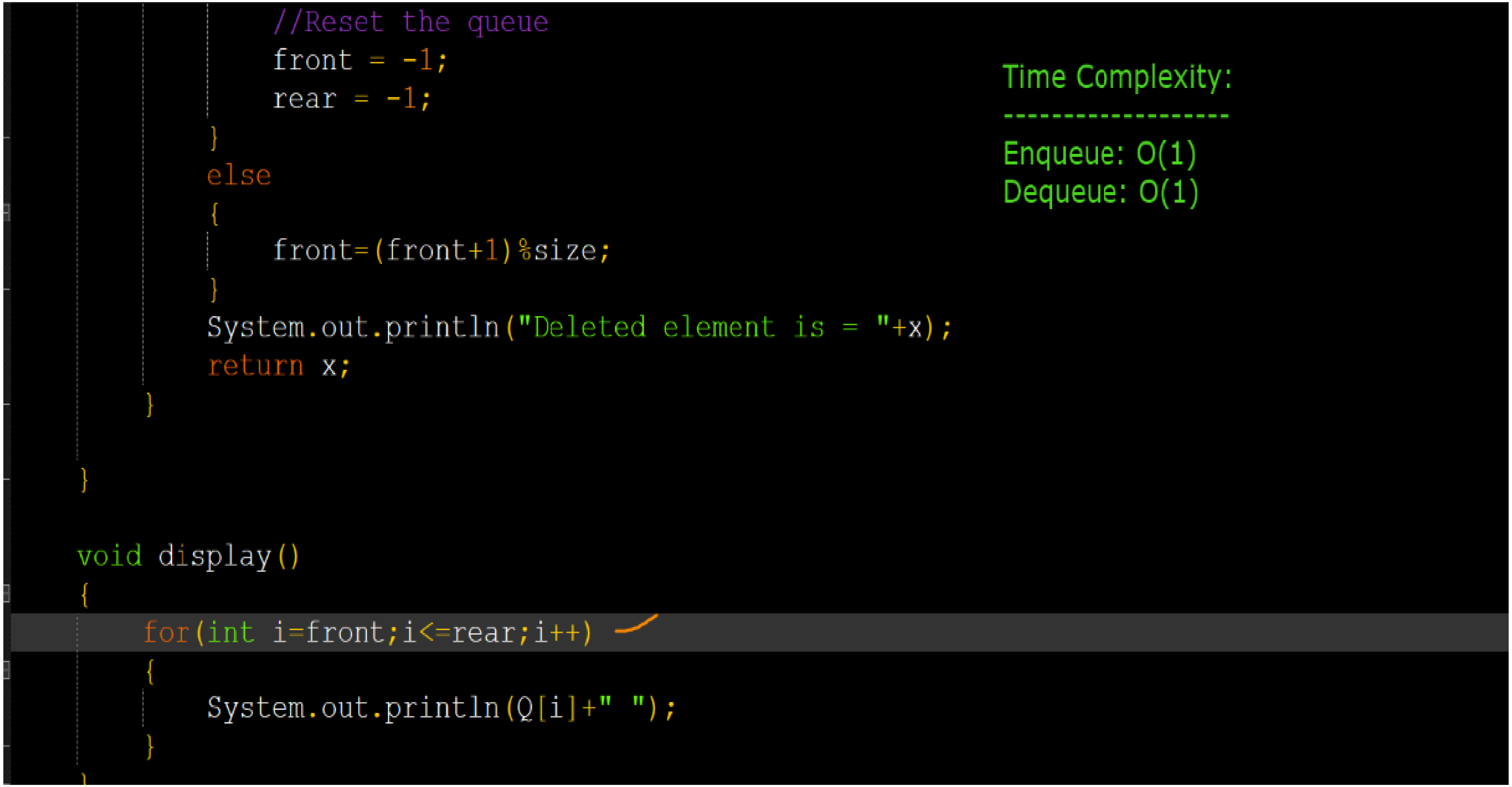


front = (front+1)%(size)

rear = (rear +1)%(size)

= (4+1)%5
=5%5
=0





```
            //Reset the queue
            front = -1;
            rear = -1;
        }
        else
        {
            front=(front+1)%size;
        }
        System.out.println("Deleted element is = "+x);
        return x;
    }

}

void display()
{
    for(int i=front;i<=rear;i++)
    {
        System.out.println(Q[i]+" ");
    }
```

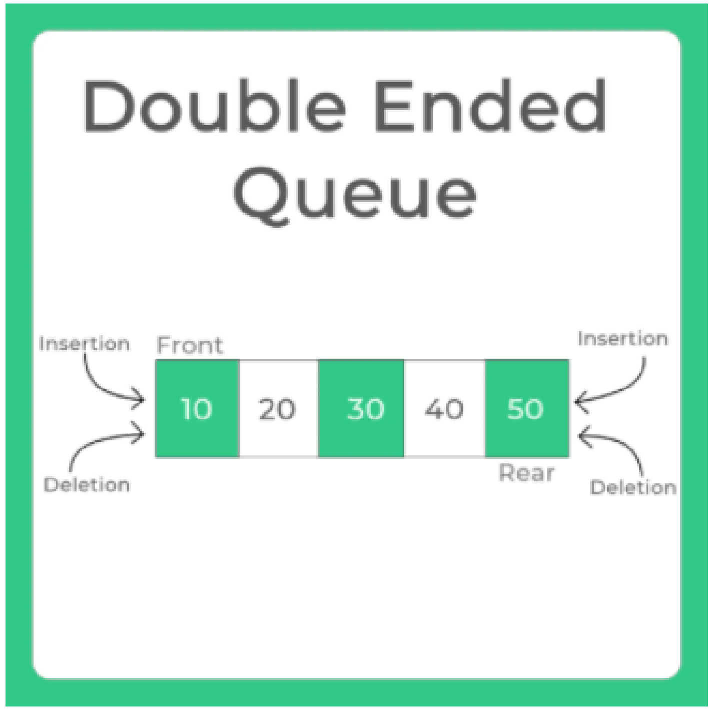
Time Complexity:

Enqueue: O(1)
Dequeue: O(1)

Double Ended Queue

Double ended queue are also known as deque. In this type of queue insertion and deletion of an element can take place at both the ends. Further deque is divided into two types:-

- Input Restricted Deque :- In this, input is blocked at a single end but allows deletion at both the ends.
- Output Restricted Deque :- In this, output is blocked at a single end but allows insertion at both the ends.

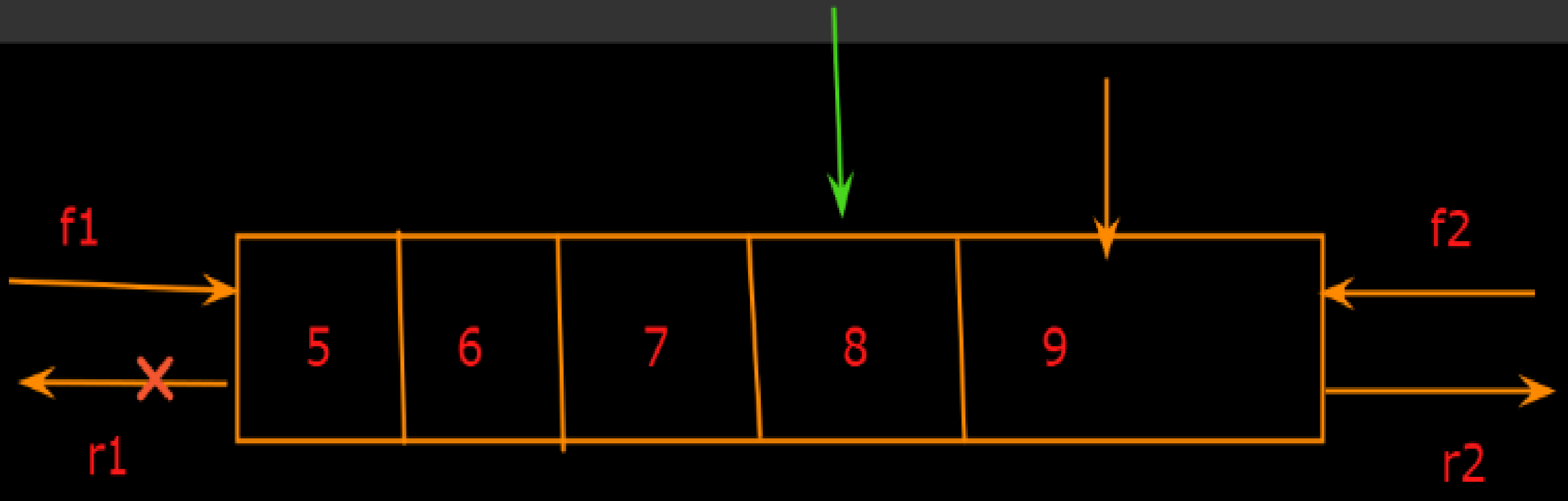


Applications of Double Ended Queue

The applications of double ended queue are:-

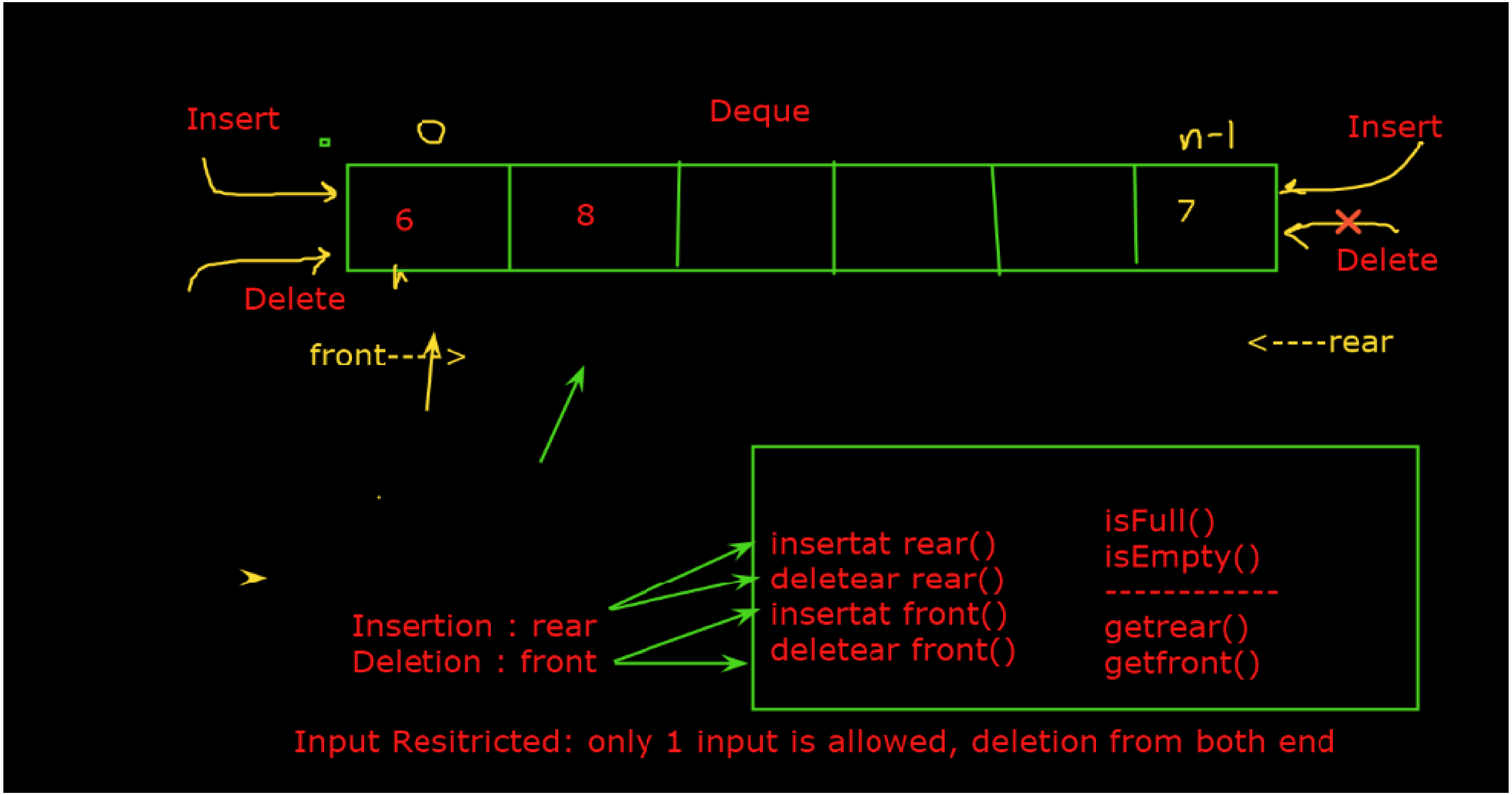
- To execute undo and redo operation.
- For implementing stacks.
- Storing the history of web browsers.

```
insertfront()
insertrear()
----------------
deletefront()
deleterear()
```



((front == 0 && rear == size-1) || front == rear + 1) :Deque full

front == -1 || rear == -1 : Deque is empty

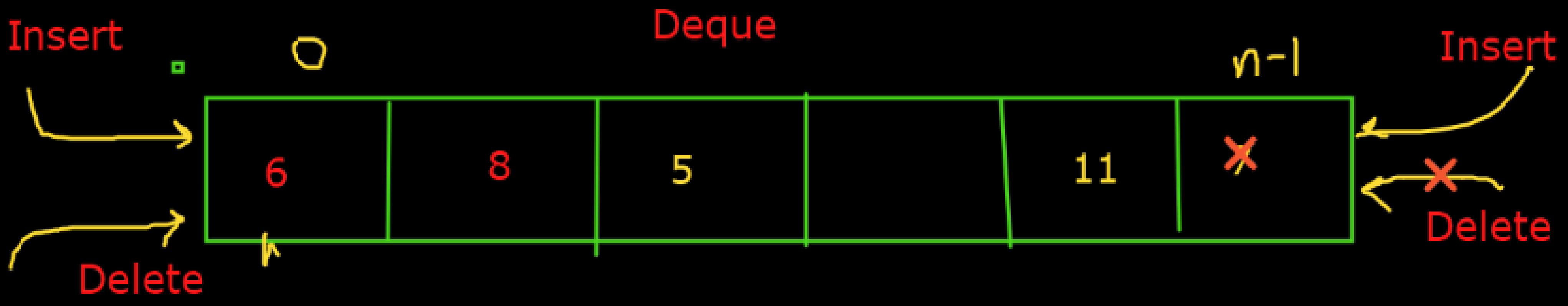


Insertion : rear
Deletion : front

insertat rear()
deletear rear()
insertat front()
deletear front()

isFull()
isEmpty()
----
getrear()
getfront()

Input Resitricted: only 1 input is allowed, deletion from both end



F=1
R=2

insertrear(6)
insertrear(8)
deletefront()
insertreat(5)
insert front(7)
insert front(11)
deleterear()
deletefront()

Algorithms & Data Structure
Searching & Sorting

Kiran Waghmare

Searching in Arrays

- **Searching:** It is used to find out the location of the data item if it exists in the given collection of data items.

  E.g. We have linear array A as below:

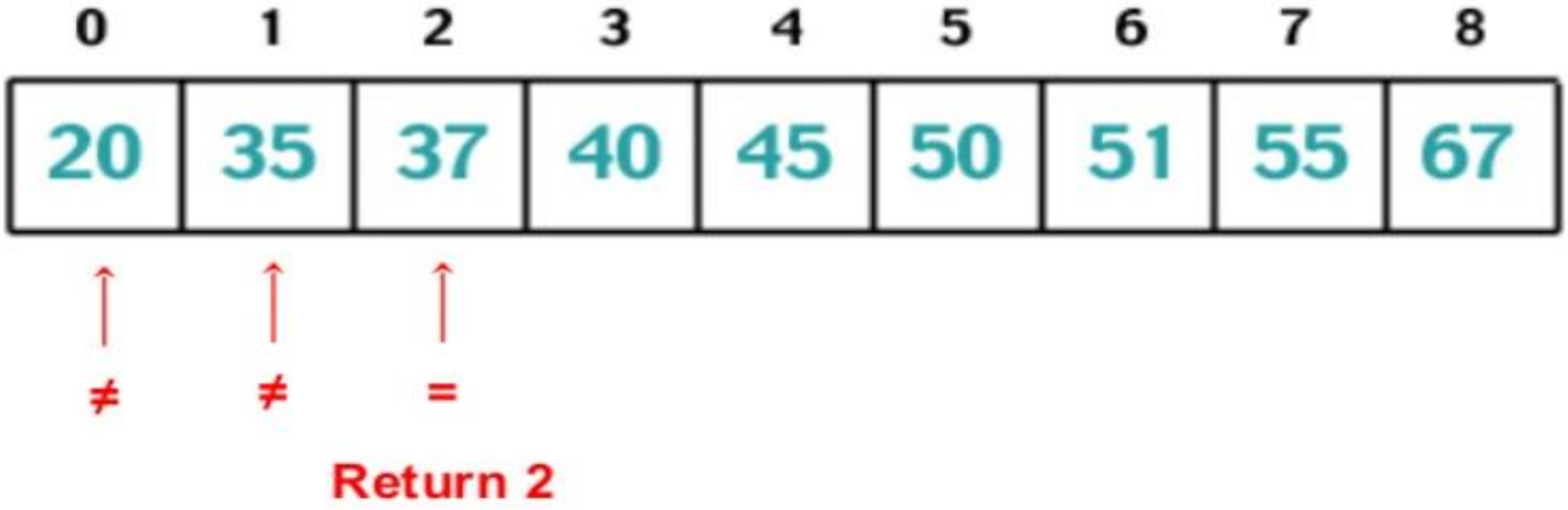
1	2	3	4	5
15	50	35	20	25

  Suppose item to be searched is 20. We will start from beginning and will compare 20 with each element. This process will continue until element is found or array is finished. Here:

  1) Compare 20 with 15
     20 # 15, go to next element.
  2) Compare 20 with 50
     20 # 50, go to next element.
  3) Compare 20 with 35
     20 #35, go to next element.
  4) Compare 20 with 20
     20 = 20, so 20 is found and its location is 4.

Linear Search

- Find 37?



Linear Search

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to find an element with a value of ITEM using sequential search.

```
1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop
```

Program 3

Problem: Given an array arr[] of n elements, write a function to search a given element x in arr[].

Examples :

```
Input : arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}
          x = 110;
Output : 6
Element x is present at index 6

Input : arr[] = {10, 20, 80, 30, 60, 50,
                     110, 100, 130, 170}
           x = 175;
Output : -1
Element x is not present in arr[].
```

The time complexity of the above algorithm is O(n).

Linear search is rarely used practically because other search algorithms such as the binary search algorithm and hash tables allow significantly faster-searching comparison to Linear search.

Improve Linear Search Worst-Case Complexity

if element Found at last O(n) to O(1)
if element Not found O(n) to O(n/2)

Binary Search

- Find 37?
  1. Sort Array.

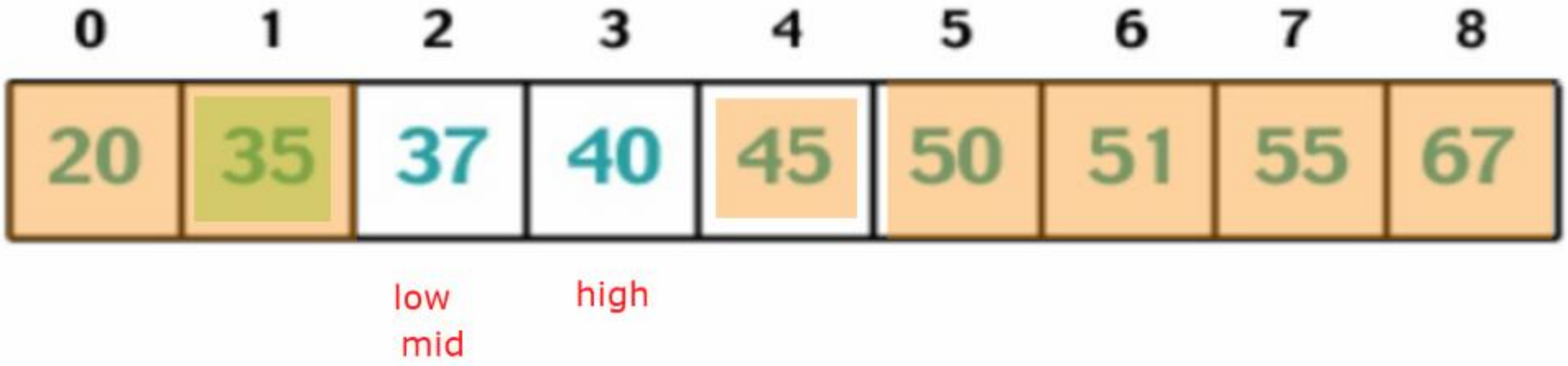
0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67

Binary Search

low	high	mid
0	8	4 → 1
0	3	1 → 2
2	3	2 → 3

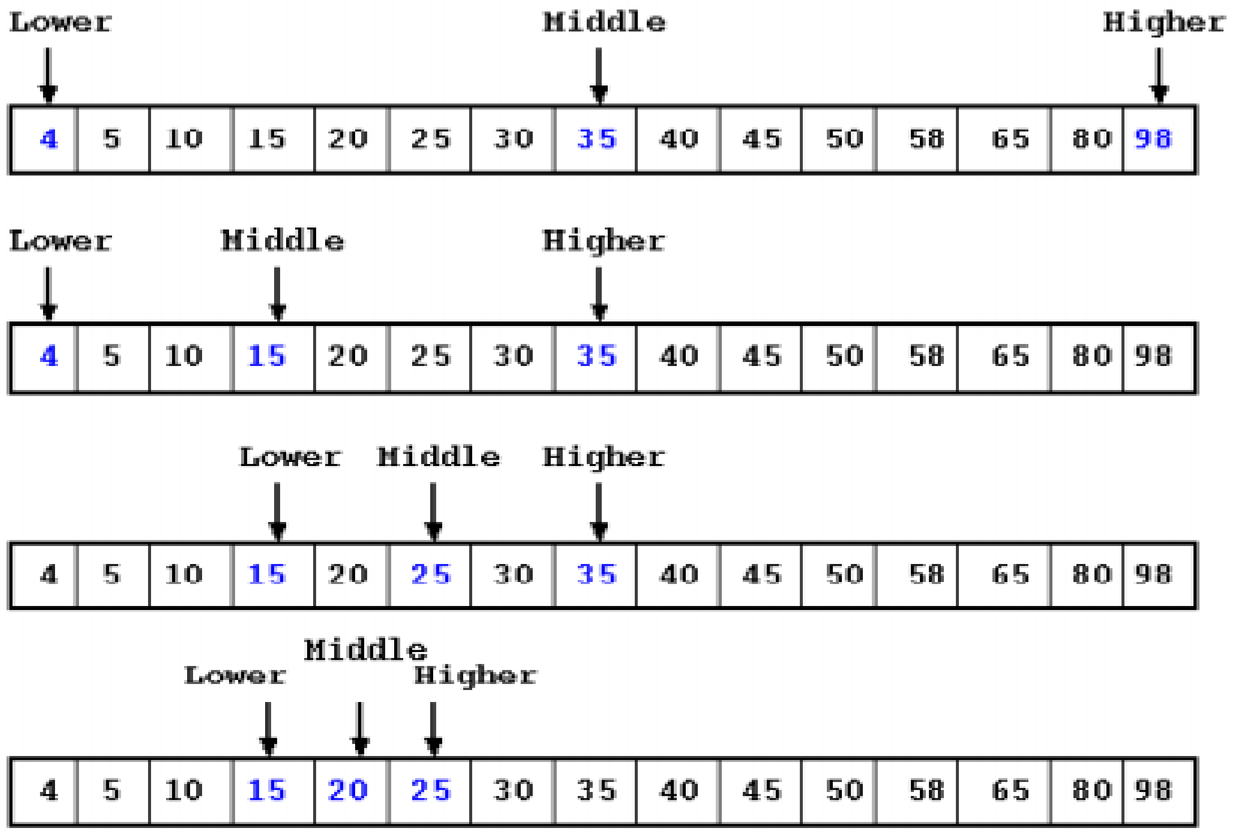
- Find 37?

1. Sort Array.

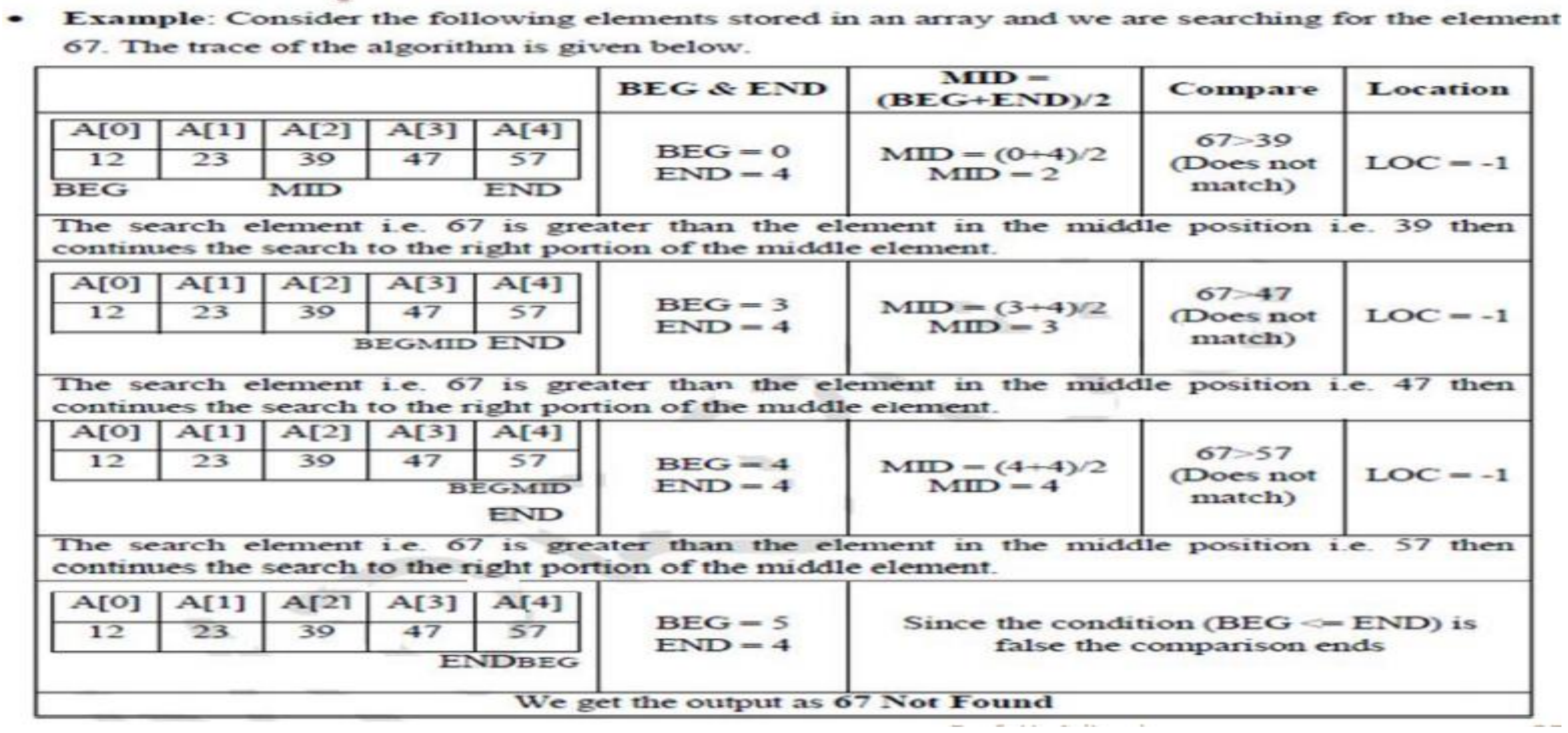


Binary Search

- The binary search algorithm can be used with only sorted list of elements.
- Binary Search first divides a large array into two smaller sub-arrays and then recursively operate the sub-arrays.
- Binary Search basically reduces the search space to half at each step



Binary Search



- **Example**: Consider the following elements stored in an array and we are searching for the element 67. The trace of the algorithm is given below.

	BEG & END	**MID = (BEG+END)/2**	**Compare**	**Location**

A[0] A[1] A[2] A[3] A[4]
12 23 39 47 57
BEG MID END
	BEG = 0
	END = 4
		MID = (0+4)/2
		MID = 2
			67>39
			(Does not match)
				LOC = -1

The search element i.e. 67 is greater than the element in the middle position i.e. 39 then continues the search to the right portion of the middle element.				

A[0] A[1] A[2] A[3] A[4]
12 23 39 47 57
BEGMID END
	BEG = 3
	END = 4
		MID = (3+4)/2
		MID = 3
			67>47
			(Does not match)
				LOC = -1

The search element i.e. 67 is greater than the element in the middle position i.e. 47 then continues the search to the right portion of the middle element.				

A[0] A[1] A[2] A[3] A[4]
12 23 39 47 57
BEGMID
END
	BEG = 4
	END = 4
		MID = (4+4)/2
		MID = 4
			67>57
			(Does not match)
				LOC = -1

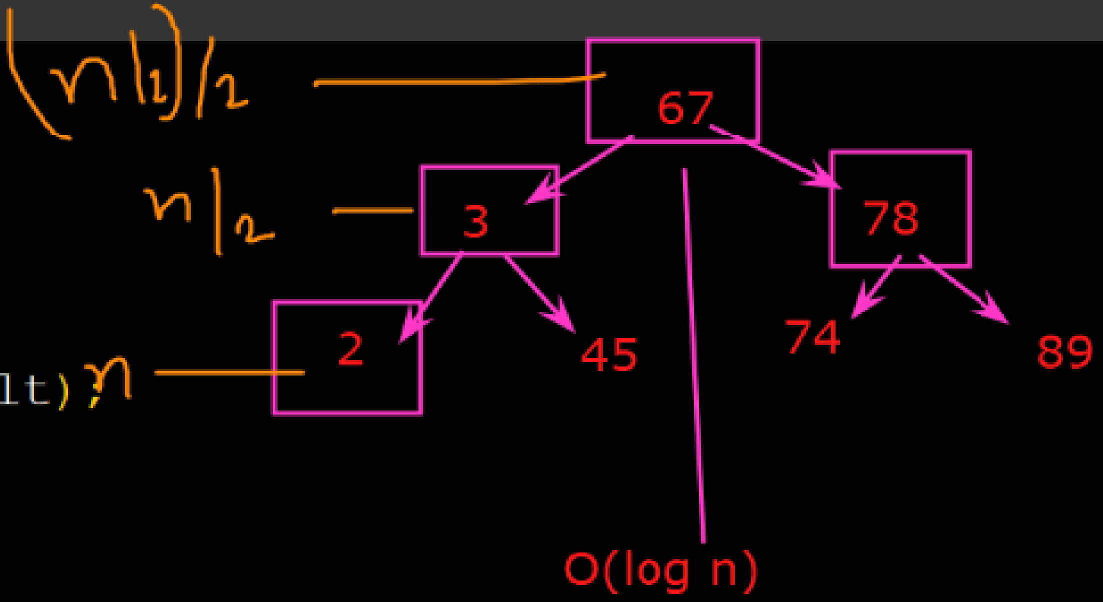
The search element i.e. 67 is greater than the element in the middle position i.e. 57 then continues the search to the right portion of the middle element.				

A[0] A[1] A[2] A[3] A[4]
12 23 39 47 57
ENDBEG
	BEG = 5
	END = 4
		Since the condition (BEG <= END) is false the comparison ends		

We get the output as **67 Not Found**				

```java
public static void main(String args[]){

    int arr[]={2,31,45,67,74,78,89};
    int x =2;
    int n = arr.length;
    int result = search(arr, x,0,n-1);
    if(result == -1)
        System.out.println("Not found!");
    else
        System.out.println("Found! "+result);
}
```



Root node: <comp   Min (Best case)   O(1)
Leaf node: > comp   Max (Worst case)   O(log n)
Height of tree---> time complexity

Sorting Techniques

Introduction

- Sorting is among the most basic problems in algorithm design.

- We are given a sequence of items, each associated with a given key value.
- And the problem is to rearrange the items so that they are in an increasing(or decreasing) order by key.

- The methods of sorting can be divided into two categories:
  - Internal Sorting
  - External Sorting

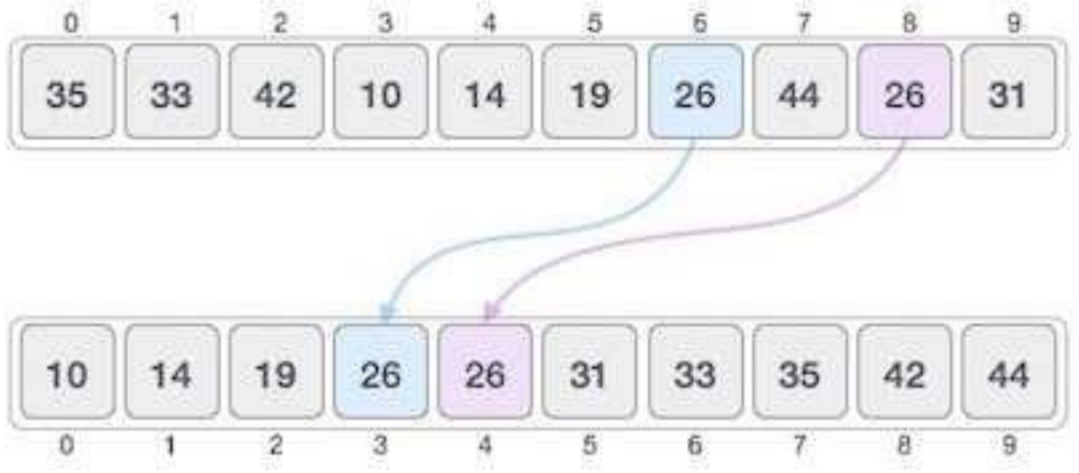
- Internal Sorting
    - ✓ If all the data that is to be sorted can be adjusted at a time in main memory, then internal sorting methods are used

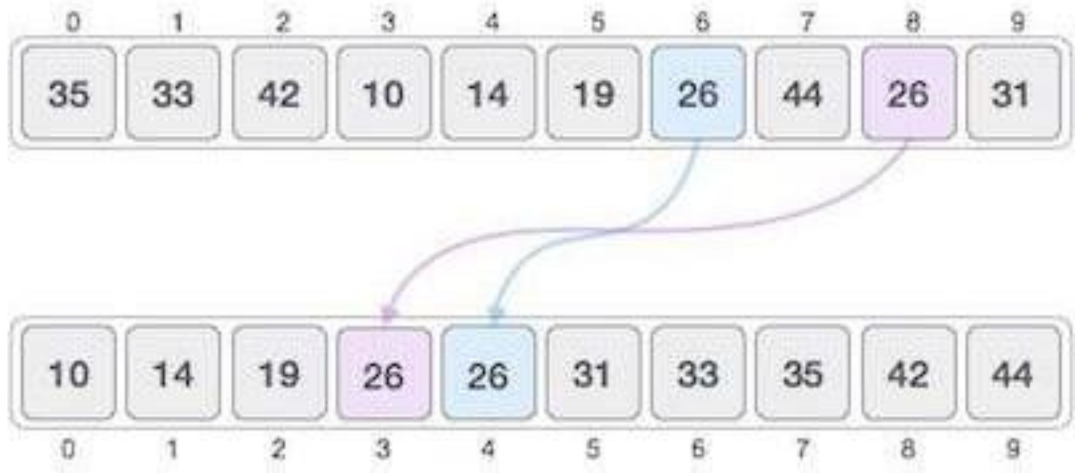
- External Sorting
    - ✓ When the data to be sorted can't be accommodated in the memory at the same time and some has to be kept in auxiliary memory, then external sorting methods are used.

❖ NOTE: We will only consider internal sorting

Stable and Not Stable Sorting

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

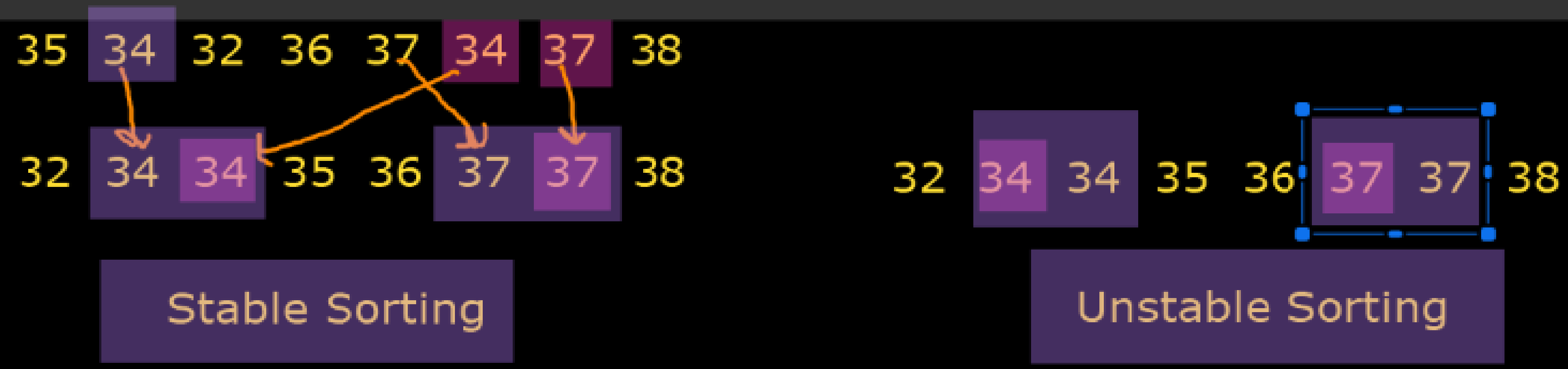




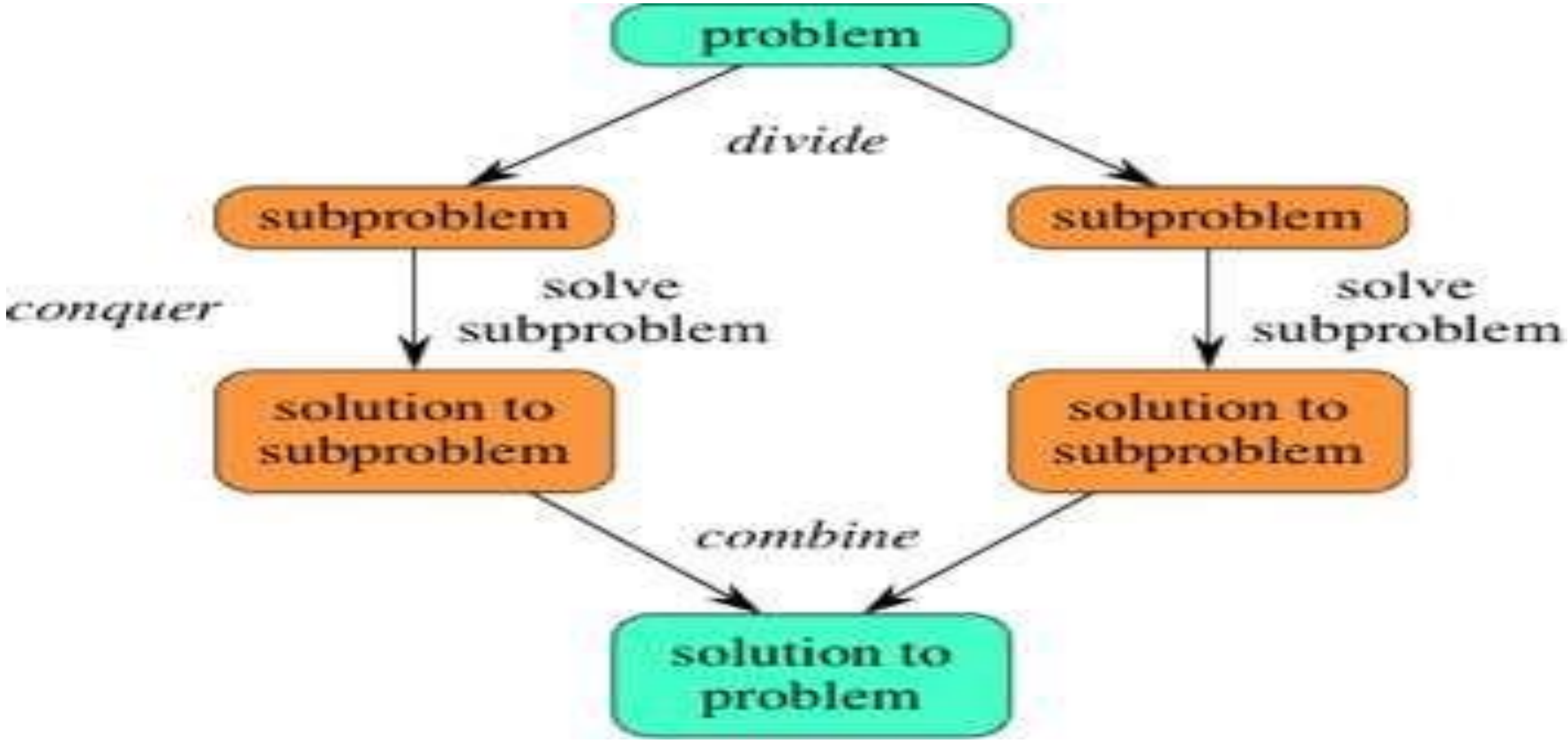
- If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called unstable sorting.

```
    When data is to be sorted can be adjusted at a time in the main memory i
    Internal Sorting.

Stable and not stable sorting:
-------------------------------
```



Divide-and-conquer



Merge Sort

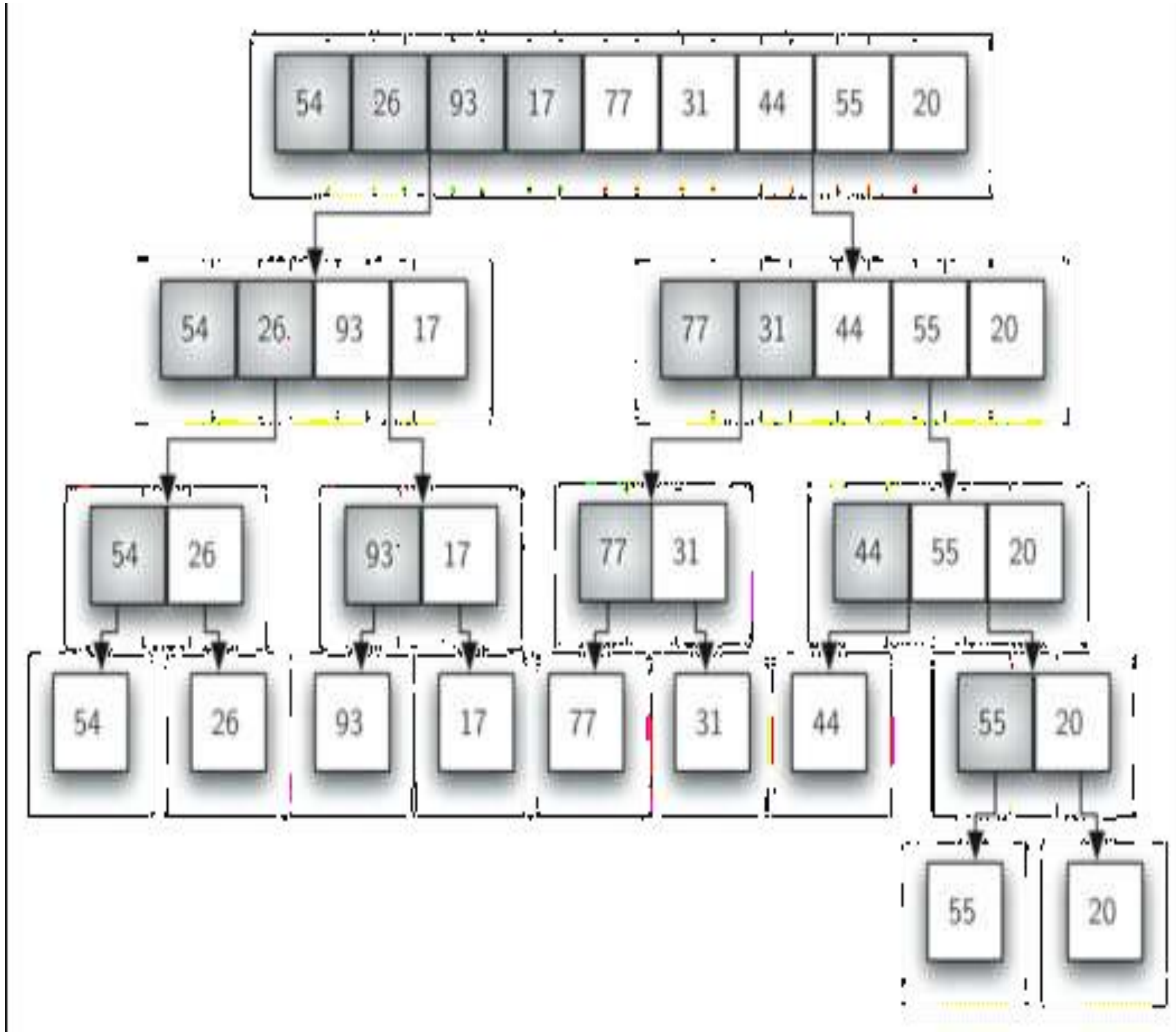
- Merge sort is a sorting technique based on divide and conquer technique.
- Merge sort first divides the array into equal halves and then combines them in a sorted manner.
- With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

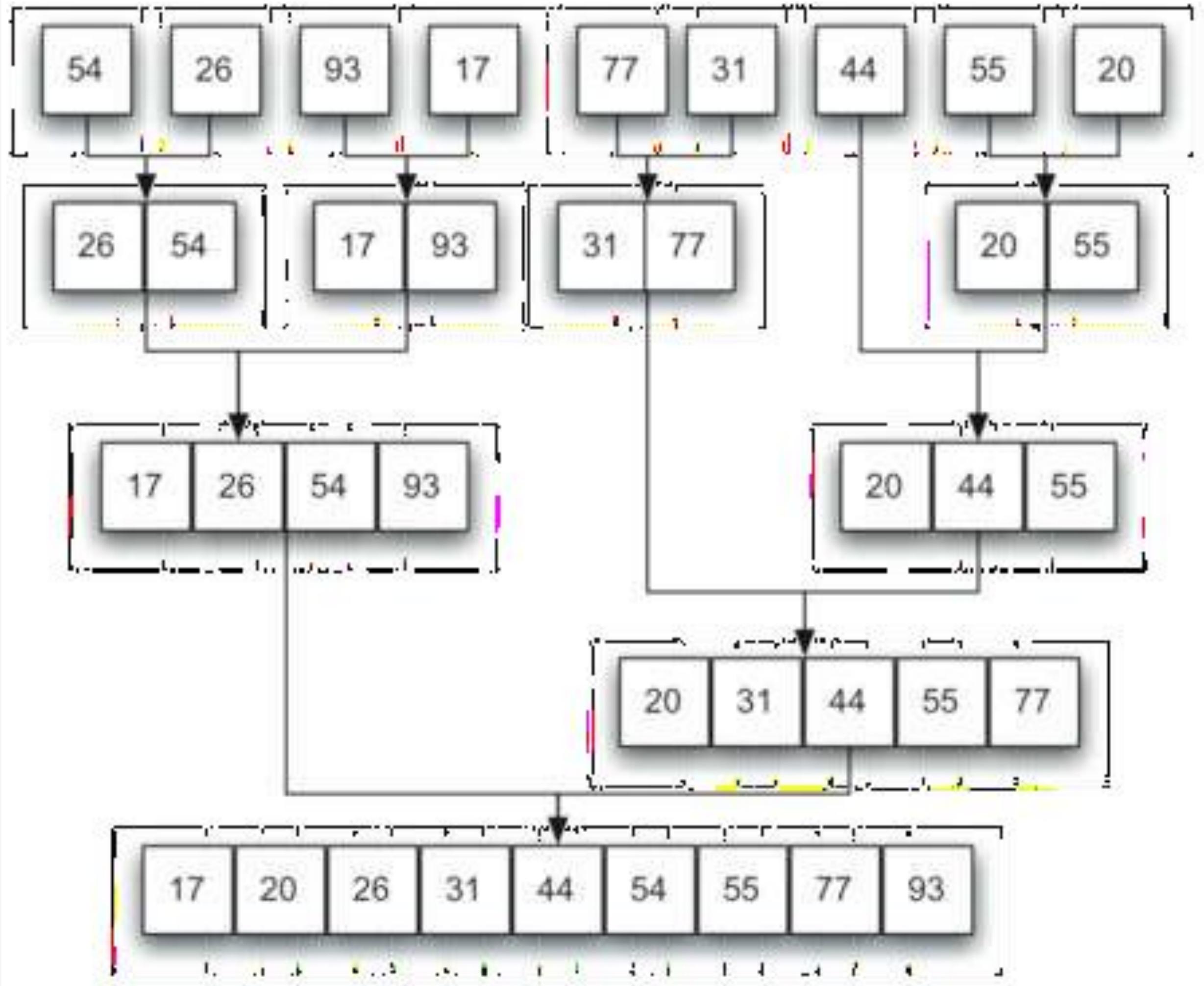
Merge Sort

- Because we're using divide-and-conquer to sort, we need to decide what our sub problems are going to be.

- Full Problem: Sort an entire Array
- Sub Problem: Sort a sub array

- Lets assume array[p..r] denotes this subarray of array.
- For an array of n elements, we say the original problem is to sort array[0..n-1]

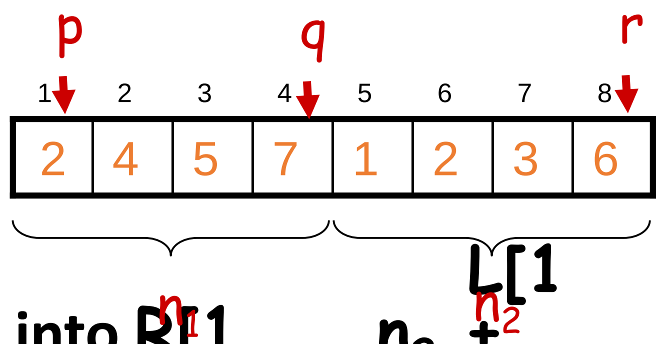

54 26 93 17 77 31 44 55 20
54 26 93 17
77 31 44 55 20
54 26
93 17
77 31
44 55 20
54
26
93
17
77
31
44
55 20
55
20

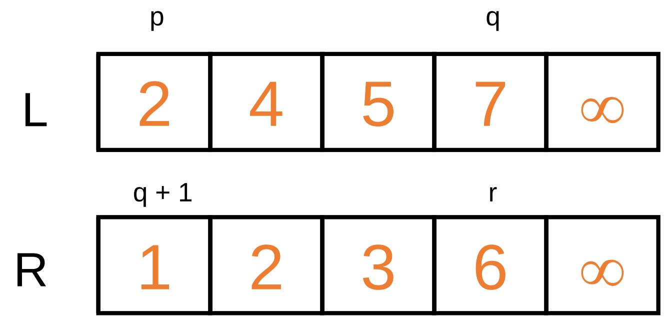

54 26 93 17 77 31 44 55 20
26 54
17 93
31 77
20 55
17 26 54 93
20 44 55
20 31 44 55 77
17 20 26 31 44 54 55 77 93

Merge - Pseudocode

Alg.: **MERGE(A, p, q, r)**

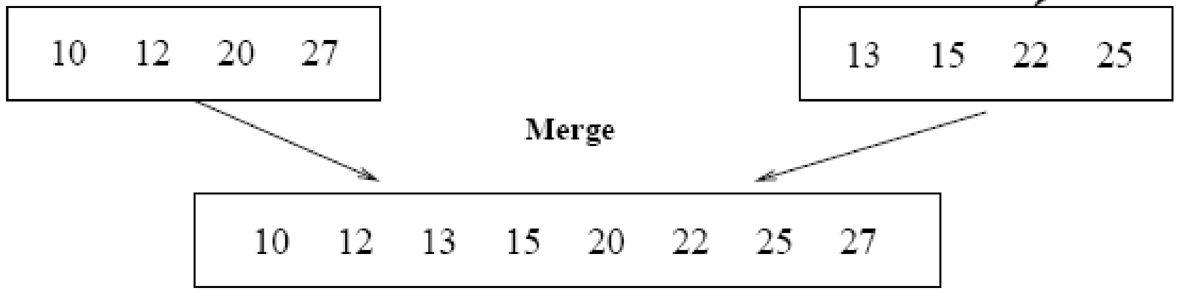
```
1.  Compute $n_1$ and $n_2$
2.  Copy the first $n_1$ elements into L[1 . . $n_1 + 1$] and the next $n_2$ elements into R[1 . . $n_2 + 1$]
3.  L[$n_1 + 1$] ← ∞;   R[$n_2 + 1$] ← ∞
4.  i ← 1;   j ← 1
5.  for k ← p to r
6.      do if L[ i ] ≤ R[ j ]
7.          then A[k] ← L[ i ]
8.               i ← i + 1
9.          else A[k] ← R[ j ]
10.              j ← j + 1
```





Running Time of Merge (assume last for loop)

- Initialization (copying into temporary arrays):
  - $\Theta(n_1 + n_2) = \Theta(n)$
- Adding the elements to the final array:
  - n iterations, each taking constant time $\Rightarrow \Theta(n)$
- Total time for Merge:
  - $\Theta(n)$



Merge Sort - Discussion

- **Running time insensitive of the input**

- **Advantages:**
  - Guaranteed to run in $\Theta(n\lg n)$

- **Disadvantage**
  - Requires extra space $\approx$N

Merge Sort:

Here is the pseudocode for Merge Sort, modified to include a counter:

```
count ← 0
Merge_Sort(A, p, r)
1       if p < r
2           then  q ← ⌊(p + r)/2⌋
3                 Merge-Sort (A, p, q)
4                 Merge-Sort (A, q+1, r)
5                 Merge (A, p, q, r)
```

And here is the modified algorithm for the Merge function used by Merge Sort:

```
Merge (A, p, q, r)
1       n1 ← (q – p) + 1
2       n2 ← (r – q)
3       create arrays L[1..n1+1] and R[1..n2+1]
4       for i ← 1 to n1 do
5           L[i] ← A[(p + i) -1]
6       for j ← 1 to n2 do
7           R[j] ← A[q + j]
8       L[n1 + 1] ← ∞
9       R[n2 + 1] ← ∞
10      i ← 1
11      j ← 1
12      for k ← p to r do
12.5        count ← count + 1
13          if L[I] <= R[j]
14              then  A[k] ← L[i]
15                    i ←i + 1
16              else A[k] ← R[j]
17                    j ← j + 1
```

Analysis of merge Sort

- Divide and conquer
- Recursive
- Stable
- Not In-place
- 0(n) space complexity
- 0(nlogn) time complexity

Quick Sort

- Quick sort is one of the most popular sorting techniques.
- As the name suggests the quick sort is the fastest known sorting algorithm in practice.
- It has the best average time performance.
- It works by partitioning the array to be sorted and each partition in turn sorted recursively. Hence also called partition exchange sort.

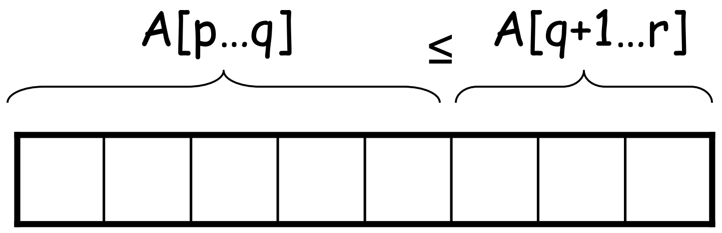
Algorithm

- Choosing a pivot
  - Topartition the list we first choose a pivot element

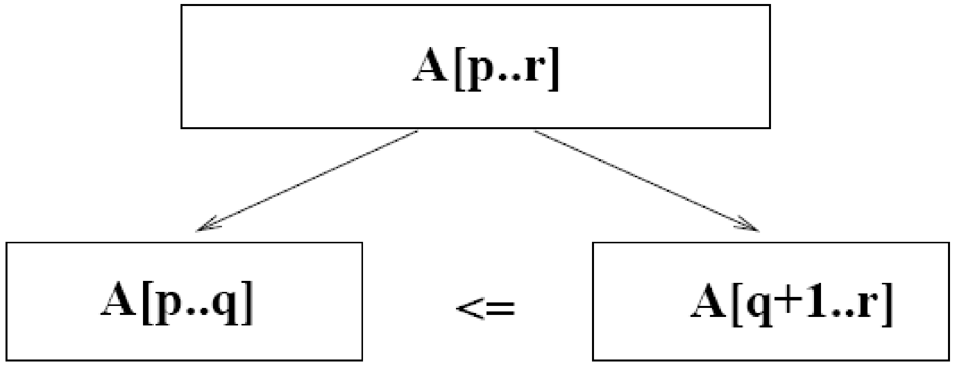
- Partitioning
  - Then we partition the elements so that all those with values less than pivot are placed on the left side and the higher vale on the right
  - Check if the current element is less than the pivot.
    - If lesser replace it with the current element and move the wall up one position
    - else move the pivot element to current element and vice versa
- Recur
  - Repeat the same partitioning step unless all elements are sorted

Quicksort

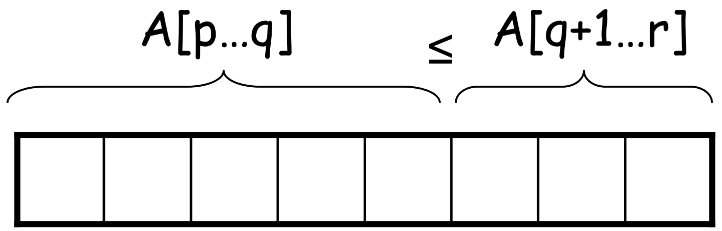
- **Sort an array *A[p…r]***



- **Divide**
    - Partition the array *A* into 2 subarrays *A*[p..q] and *A*[q+1..r], such that each element of *A*[p..q] is smaller than or equal to each element in *A*[q+1..r]
    - Need to find index *q* to partition the array



Quicksort



- **Conquer**
  - Recursively sort $A[p..q]$ and $A[q+1..r]$ using Quicksort
- **Combine**
  - Trivial: the arrays are sorted in place
  - No additional work is required to combine them
  - The entire array is now sorted

The following procedure implements quicksort:

```
QUICKSORT($A, p, r$)
1  if $p < r$
2      $q$ = PARTITION($A, p, r$)
3      QUICKSORT($A, p, q - 1$)
4      QUICKSORT($A, q + 1, r$)
```

To sort an entire array A, the initial call is QUICKSORT(A, 1, $A.length$).

Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p \,..\, r]$ in place.

```
PARTITION($A, p, r$)
1  $x = A[r]$
2  $i = p - 1$
3  for $j = p$ to $r - 1$
4      if $A[j] \le x$
5          $i = i + 1$
6          exchange $A[i]$ with $A[j]$
7  exchange $A[i + 1]$ with $A[r]$
8  return $i + 1$
```

Worst Case Partitioning

- **Worst-case partitioning**
  - One region has one element and the other has n – 1 elements
  - Maximally unbalanced

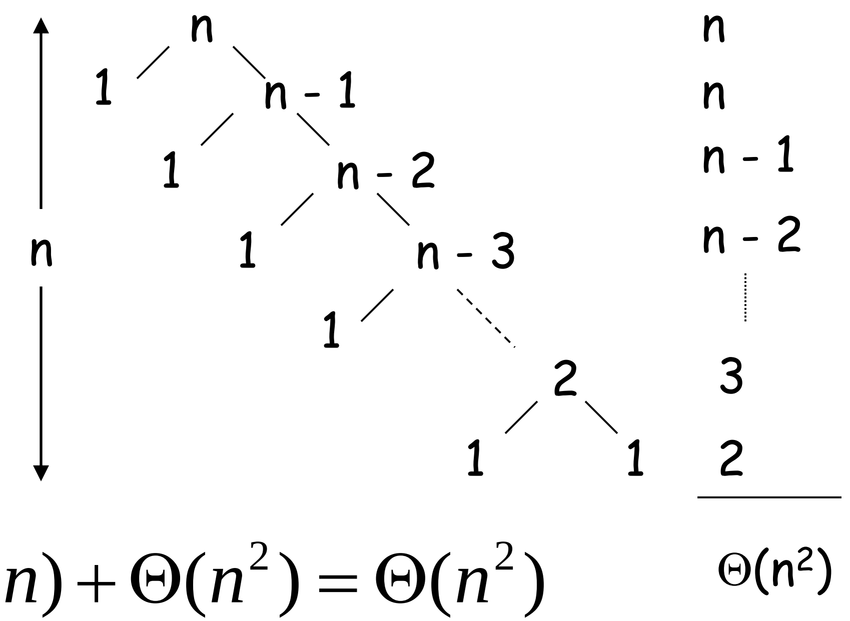
- **Recurrence: q=1**

T(n) = T(1) + T(n – 1) + n,

T(1) = Θ(1)

T(n) = T(n – 1) + n

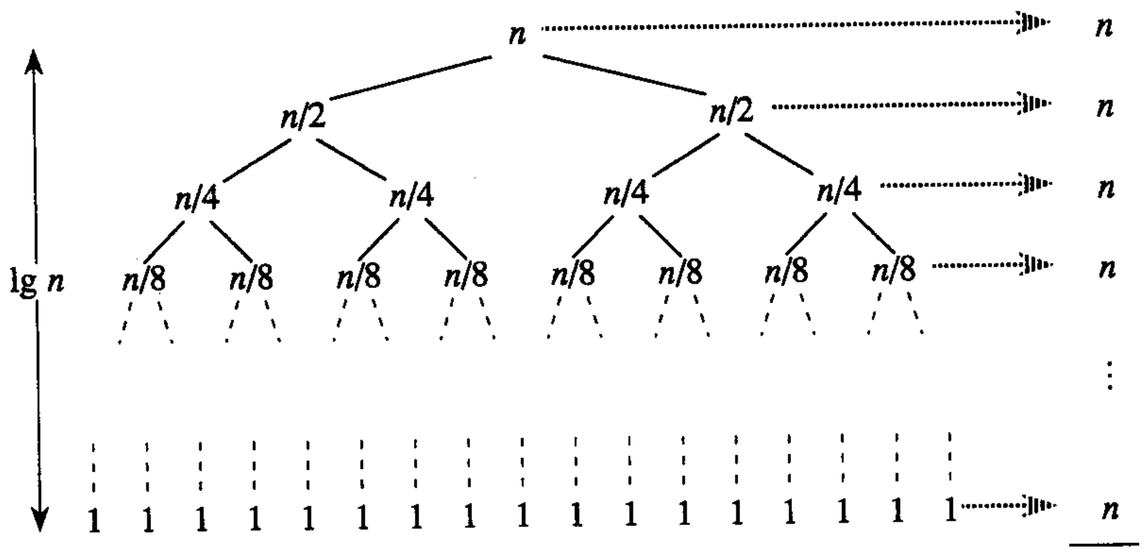
$$= n + \left(\sum_{k=1}^{n} k \right) - 1 = \Theta(n) + \Theta(n^2) = \Theta(n^2)$$



When does the worst case happen?

Best Case Partitioning

- **Best-case partitioning**
  - Partitioning produces two regions of size $n/2$



Analysis of QuickSort

- Best case
  - The best case analysis assumes that the pivot is always in the middle
  - To simplify the math, we assume that the two sublists are each exactly half the size of the original T(N)=T(N/2)+T(N/2)….+1 leads to T(N)=O(nlogn)

- Average case
  - T(N)=O(nlogn)

- Worst case
  - When we pick minimum or maximum as pivot then we have to go through each and every element so
  - T(N) = O(n^2)

BUBBLE SORT

- In bubble sort, each element is compared with its adjacent element.
- We begin with the 0th element and compare it with the 1st element.
- If it is found to be greater than the 1st element, then they are interchanged.
- In this way all the elements are compared (excluding last) with their next element and are interchanged if required
- On completing the first iteration, largest element gets placed at the last position. Similarly in second iteration second largest element gets placed at the second last position and so on.

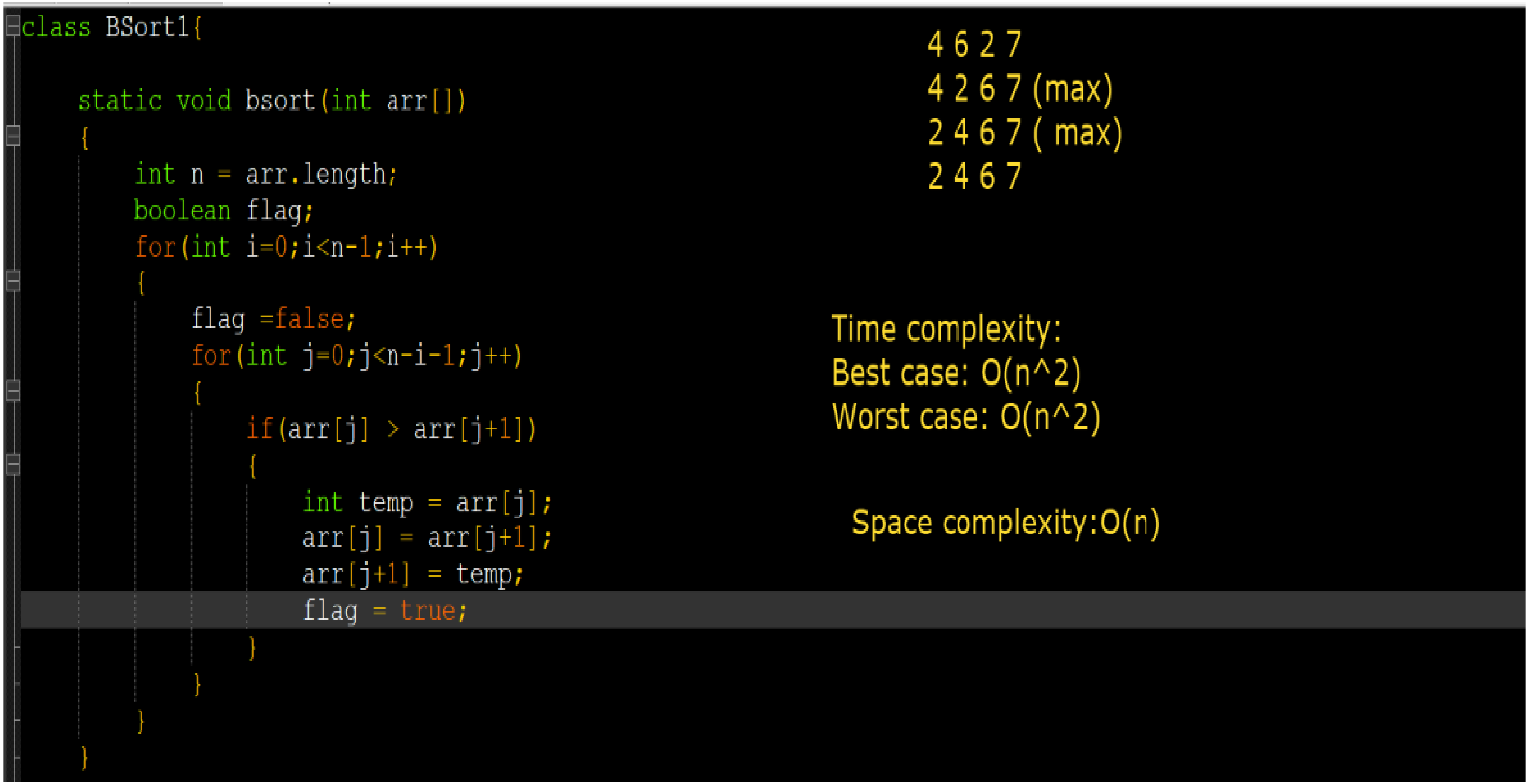
TIME COMPLEXITY

- The time complexity for bubble sort is calculated in terms of the number of comparisons f(n) (or of number of loops)
- Here two loops(outer loop and inner loop) iterates(or repeated) the comparison.
- The inner loop is iterated one less than the number of elements in the list (i.e., n-1 times) and is reiterated upon every iteration of the outer loop

$$f(n) = (n - 1) + (n - 2) + \ldots\ldots + 2 + 1$$
$$= n(n - 1) = O(n2).$$

Algorithm 1: Bubble sort

```
Data: Input array A[]
Result: Sorted A[]
int i, j, k;
N = length(A);
for j = 1 to N do
    for i = 0 to N-1 do
        if A[i] > A[i+1] then
            temp = A[i];
            A[i] = A[i+1];
            A[i+1] = temp;
        end
    end
end
```



```
class BSort1{

    static void bsort(int arr[])
    {
        int n = arr.length;
        boolean flag;
        for(int i=0;i<n-1;i++)
        {
            flag =false;
            for(int j=0;j<n-i-1;j++)
            {
                if(arr[j] > arr[j+1])
                {
                    int temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                    flag = true;
                }
            }
        }
    }
```

4 6 2 7
4 2 6 7 (max)
2 4 6 7 (max)
2 4 6 7

Time complexity:
Best case: O(n^2)
Worst case: O(n^2)

Space complexity:O(n)

SELECTION SORT

- Find the least(or greatest) value in the array, swap it into the leftmost(or rightmost) component, and then forget the leftmost component, Do this repeatedly.

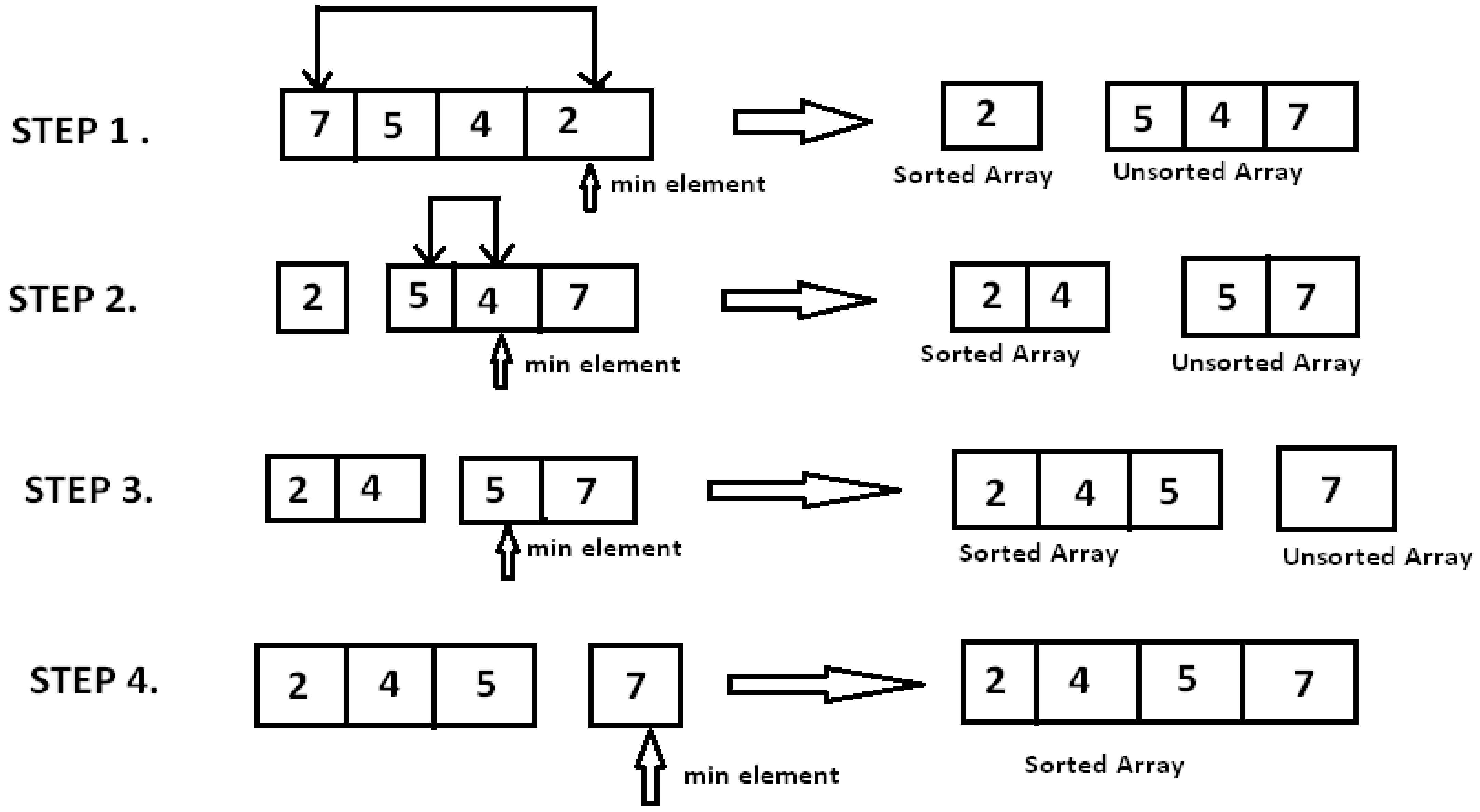
- Let a[n] be a linear array of n elements. The selection sort works as follows:

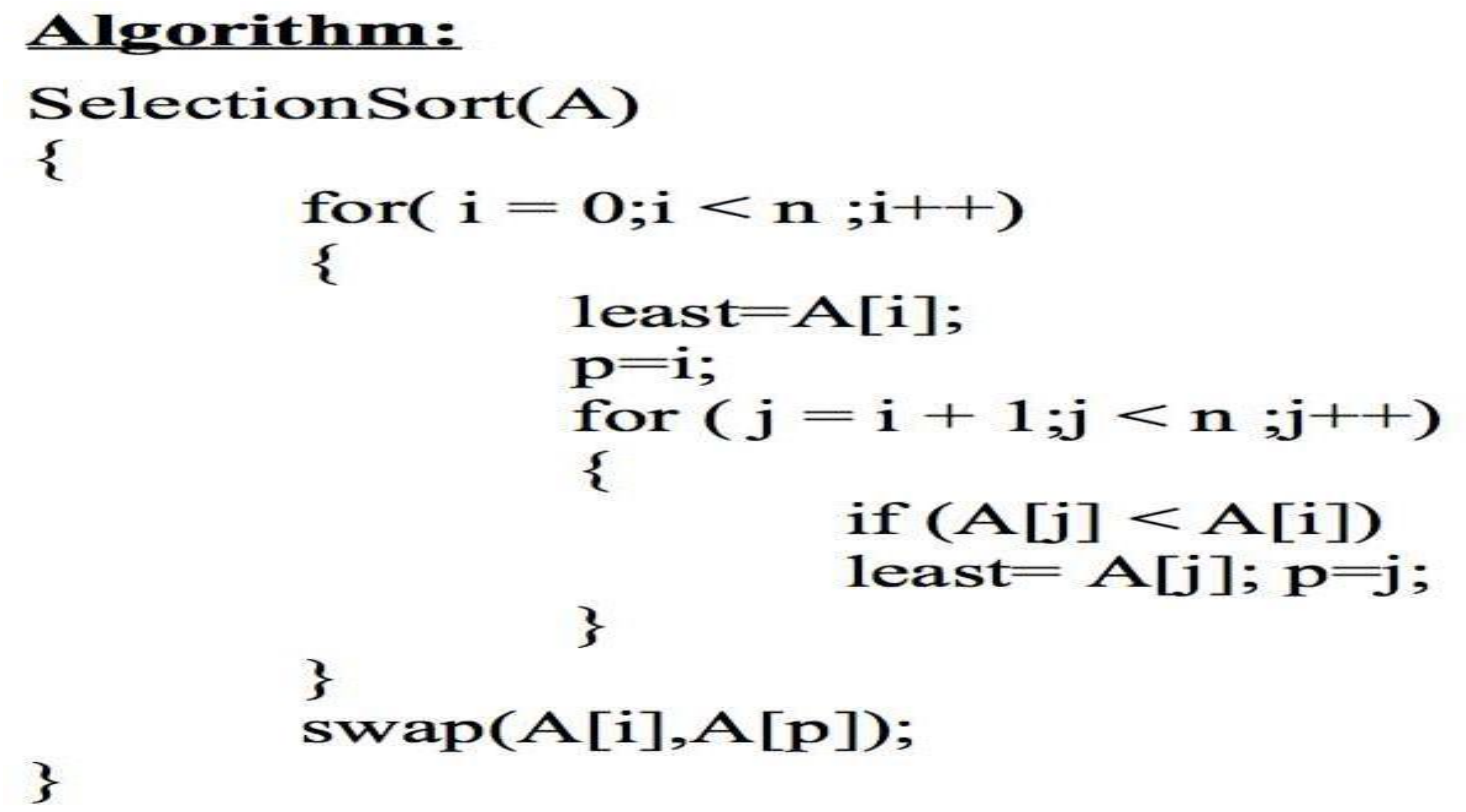
- Pass 1: Find the location loc of the smallest element in the list of n elements a[0], a[1], a[2], a[3], ...,a[n-1] and then interchange a[loc] and a[0].

- Pass 2: Find the location loc of the smallest element int the sub-list of n-1 elements a[1], a[2], a[3], ...,a[n-1] and then interchange a[loc] and a[1] such that a[0], a[1] are sorted.

- Then we will get the sorted list

  a[0]<=a[2]<=a[3]…....<=a[n-1]


STEP 1 .
7
5
4
2
min element
2
Sorted Array
5
4
7
Unsorted Array
STEP 2.
2
5
4
7
min element
2
4
Sorted Array
5
7
Unsorted Array
STEP 3.
2
4
5
7
min element
2
4
5
Sorted Array
7
Unsorted Array
STEP 4.
2
4
5
7
min element
2
4
5
7
Sorted Array



Algorithm:

```
SelectionSort(A)
{
        for( i = 0;i < n ;i++)
        {
                least=A[i];
                p=i;
                for ( j = i + 1;j < n ;j++)
                {
                        if (A[j] < A[i])
                        least= A[j]; p=j;
                }
        }
        swap(A[i],A[p]);
}
```

Time Complexity

- Inner loop executes (n-1) times when i=0, (n-2) times when i=1 and so on:
- Time complexity = (n-1) + (n-2) + (n-3) + ……….. +2+1

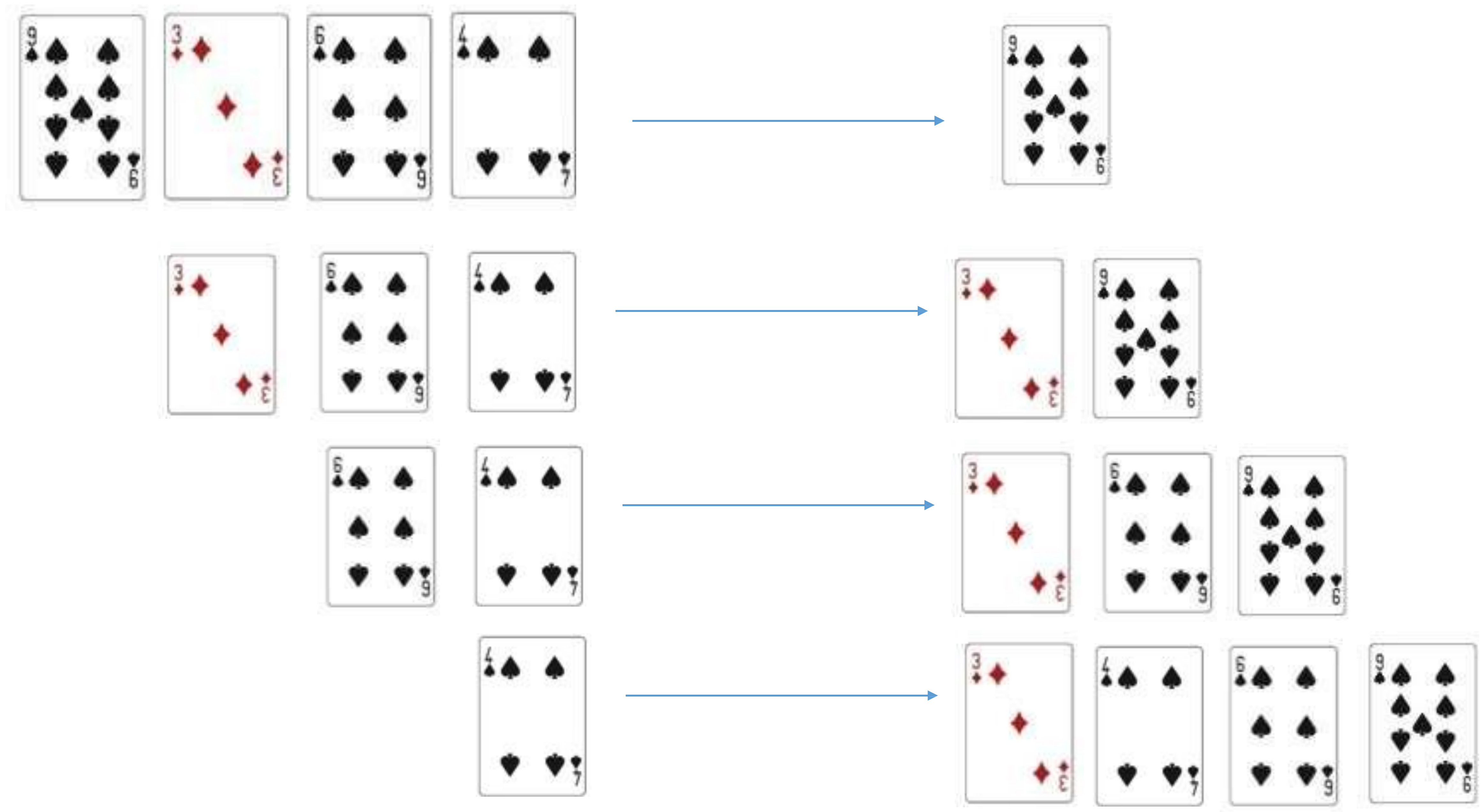
  = $O(n^2)$

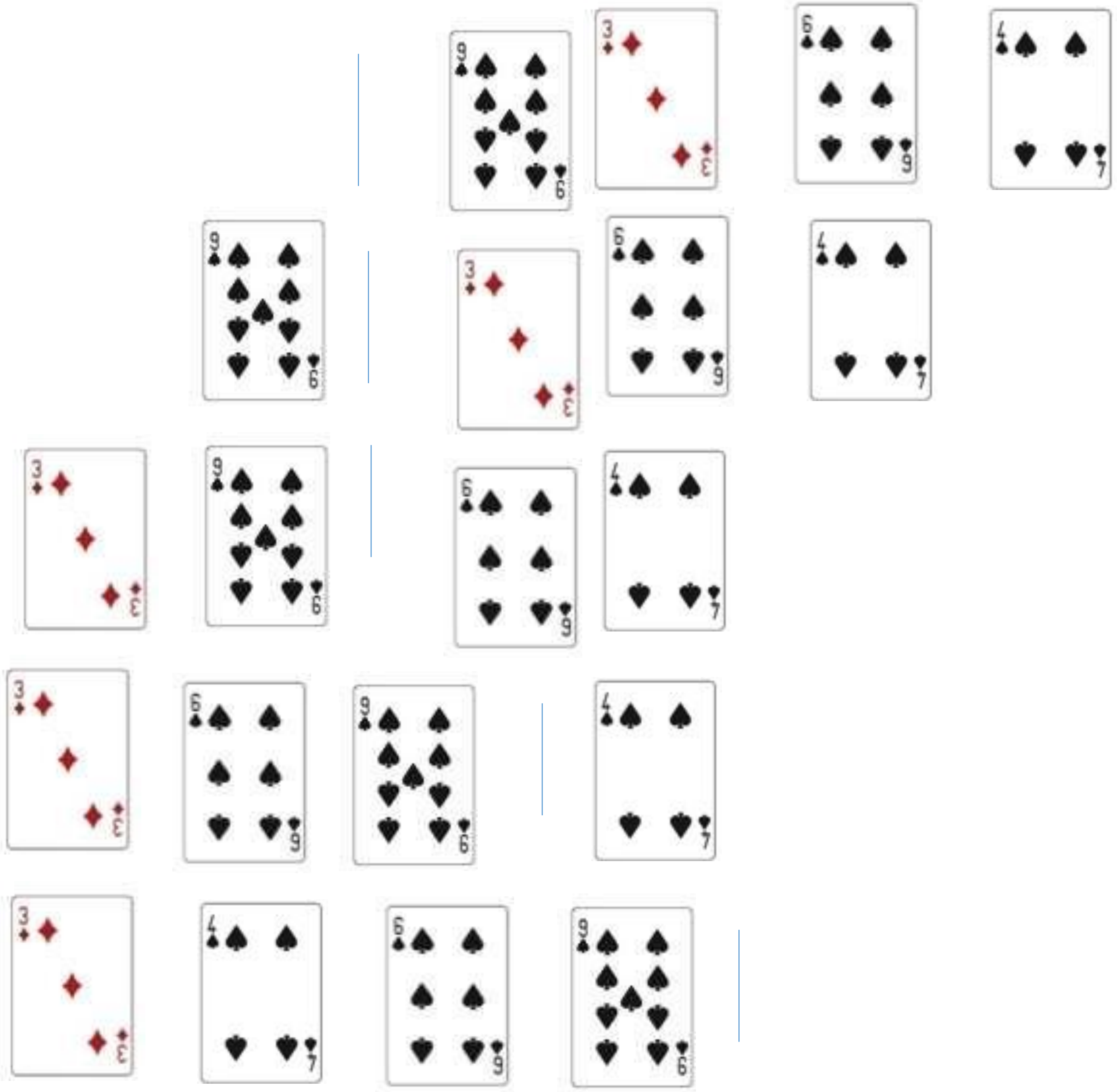
Space Complexity

- Since no extra space beside n variables is needed for sorting so
- O(n)

Insertion Sort

- Like sorting a hand of playing cards start with an empty hand and the cards facing down the table.
- Pick one card at a time from the table, and insert it into the correct position in the left hand.
- Compare it with each of the cards already in the hand, from right to left
- The cards held in the left hand are sorted.





Insertion Sort

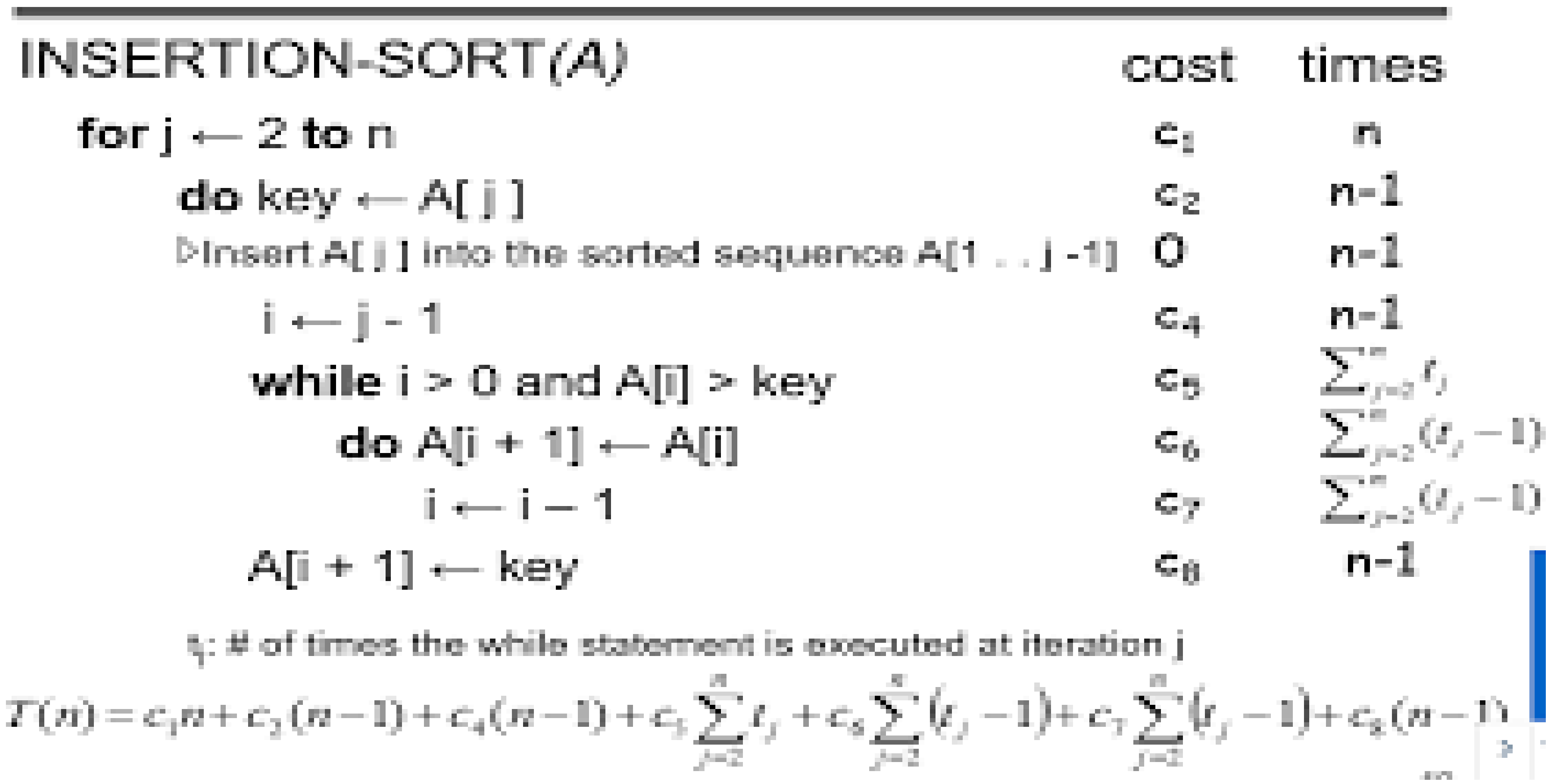
- Suppose an array a[n] with n elements. The insertion sort works asfollows:

Pass1: a[0] by itself is trivially sorted.

Pass2: a[1] is inserted either before or after a[0] so that a[0],a[1] is sorted.

Pass 3: a[2] is inserted into its proper place in a[0],a[1] that is before a[0], between a[0] and a[1], or after a[1] so that a[0],a[1],a[2] is sorted.

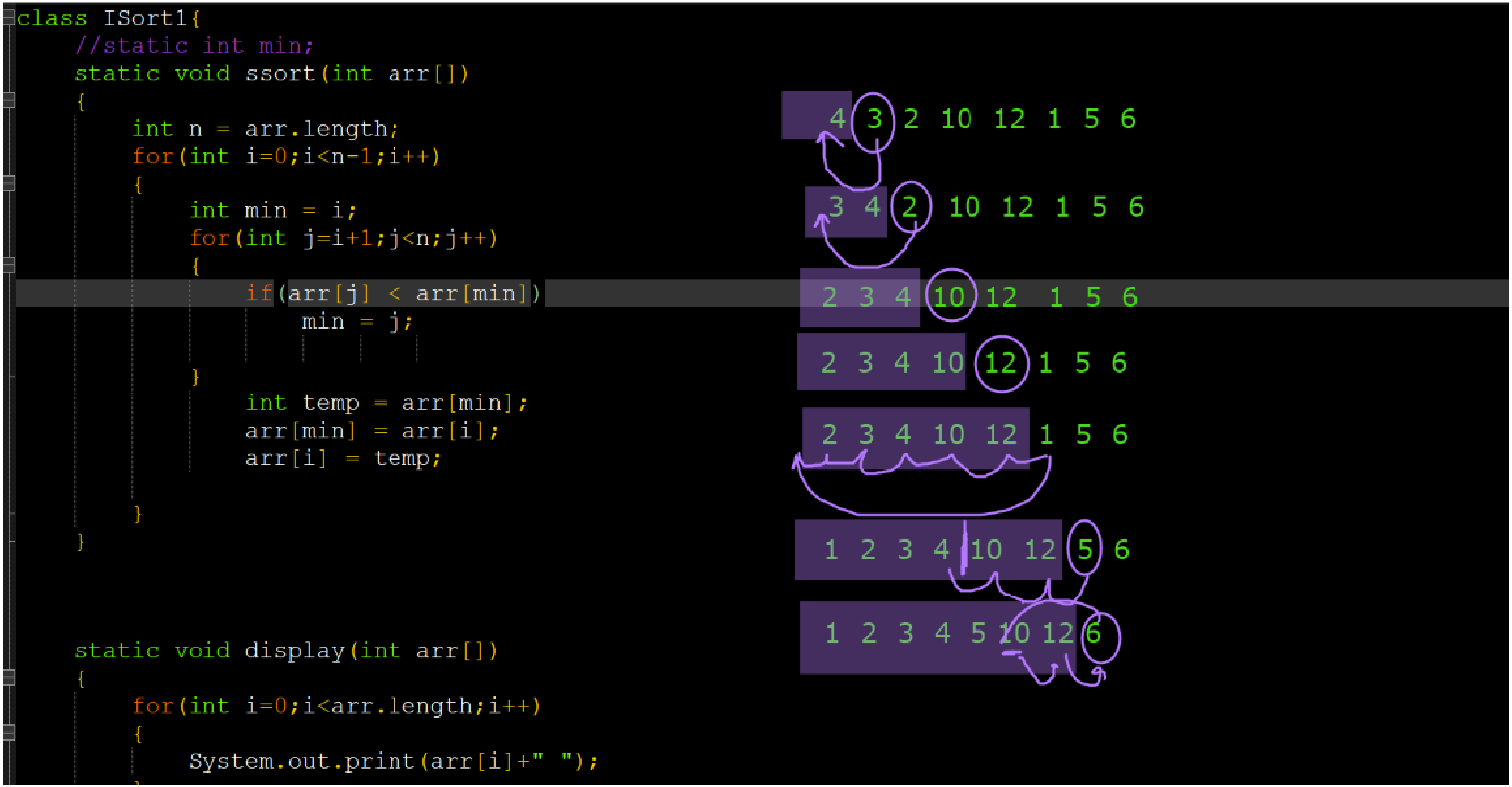
pass N: a[n-1] is inserted into its proper place in a[0],a[1],a[2],........,a[n-2] so that a[0],a[1],a[2],............,a[n-1] is sorted with nelements.



INSERTION-SORT(A)	cost	times
for $j \leftarrow 2$ **to** n	c_1	n
do key $\leftarrow A[j]$	c_2	$n-1$
▷Insert $A[j]$ into the sorted sequence $A[1 \,.\,.\, j-1]$	0	$n-1$
$i \leftarrow j - 1$	c_4	$n-1$
while $i > 0$ and $A[i] >$ key	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i \leftarrow i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] \leftarrow$ key	c_8	$n-1$

t_j: # of times the while statement is executed at iteration j

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$



```
class ISort1{
    //static int min;
    static void ssort(int arr[])
    {
        int n = arr.length;
        for(int i=0;i<n-1;i++)
        {
            int min = i;
            for(int j=i+1;j<n;j++)
            {
                if(arr[j] < arr[min])
                    min = j;

            }
                int temp = arr[min];
                arr[min] = arr[i];
                arr[i] = temp;

        }
    }


    static void display(int arr[])
    {
        for(int i=0;i<arr.length;i++)
        {
            System.out.print(arr[i]+" ");
```

Time Complexity

- Best Case:
  - If the array is all but sorted then
  - Inner Loop wont execute so only some constant time the statements will run
  - So Time complexity= O(n)

- Worst Case:
  - Array element in reverse sorted order
  - Time complexity=O(n^2)

- Space Complexity
  - Since no extra space beside n variables is needed for sorting so
  - Space Complexity = O(n)

Thanks