Applied Programming

Engr. Abdul-Rahman Mahmood

DPM, MCP, QMR(ISO9001:2000)

armahmood786@yahoo.com
alphapeeler.sf.net/pubkeys/pkey.htm
pk.linkedin.com/in/armahmood
www.twitter.com/alphapeeler
www.facebook.com/alphapeeler
abdulmahmood-sss
alphasecure
armahmood786@hotmail.com
http://alphapeeler.sf.net/me

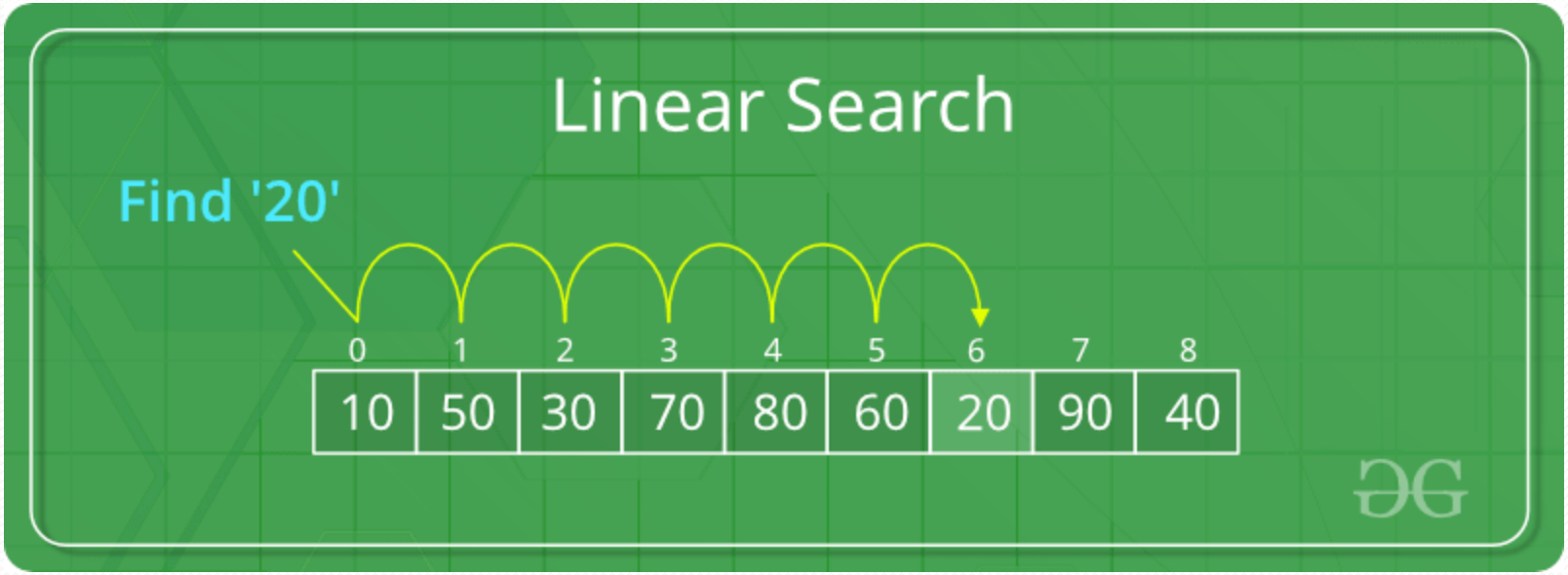
alphasecure@gmail.com
http://alphapeeler.sourceforge.net
http://alphapeeler.tumblr.com
armahmood786@jabber.org
alphapeeler@aim.com
mahmood_cubix
48660186
alphapeeler@icloud.com
http://alphapeeler.sf.net/acms/

Data structures

Searching (linear search, binary search), Stack & queues their types and operations. (types: Linear, circular, double ended and priority queues)

Linear Search

- A simple approach is to do **linear search**, i.e:
- **Algorithm:**
  - Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
  - If x matches with an element, return the index.
  - If x doesn't match with any of elements, return -1.



Ex 01

```
public class LinearSearch {
public static int search(int arr[], int x) {
    int n = arr.length;
    for(int i = 0; i < n; i++) {
        if(arr[i] == x)
            return i;
    }
    return -1;
}
public static void main(String args[]) {
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int result = search(arr, x);
    if(result == -1)
        System.out.print("Element is not present in array");
    else
        System.out.print("Element is present at index " + result);
}
}
```

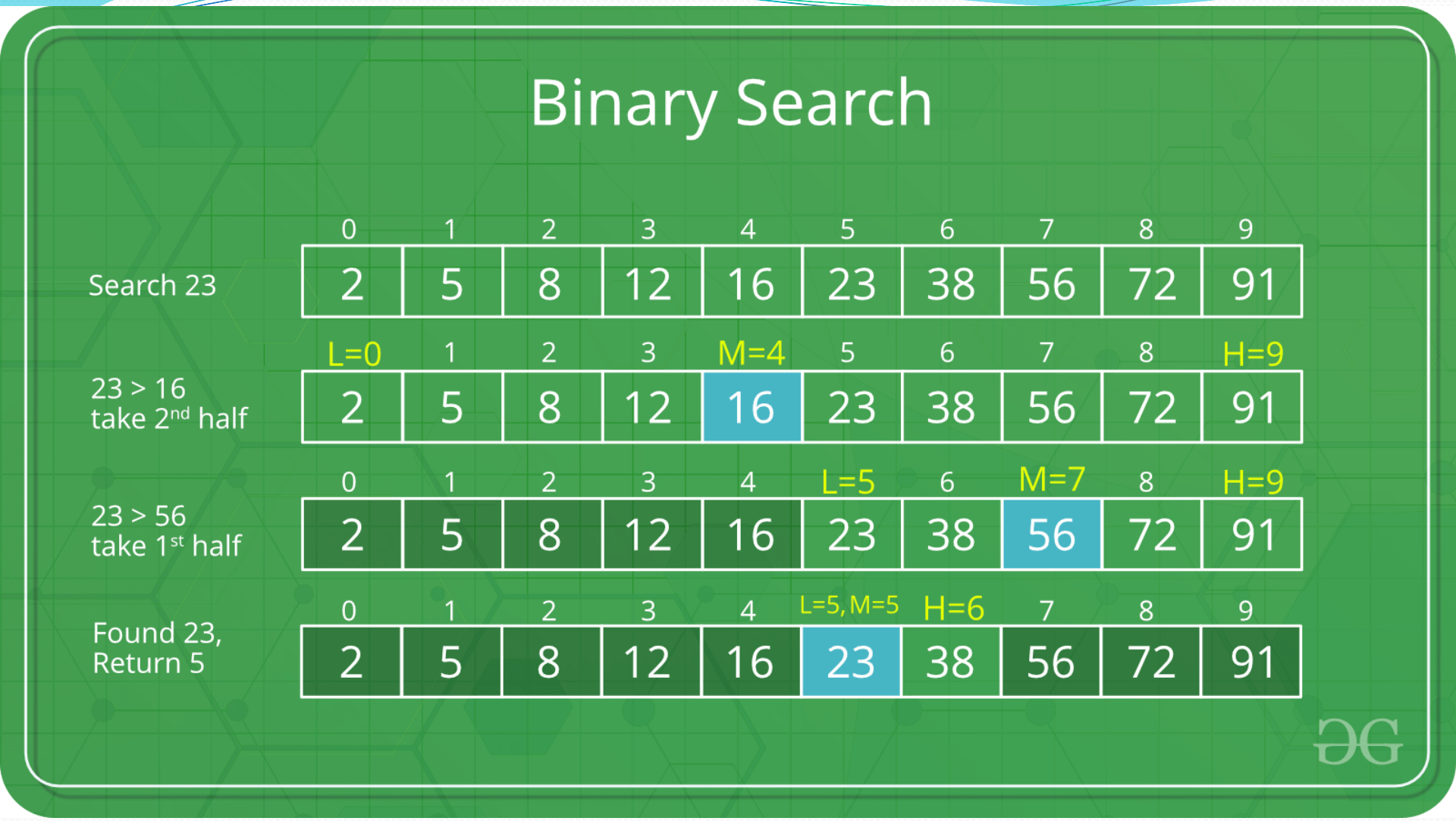
Output:

Element is present at index 3

Binary Search

- Given a sorted array arr[] of n elements, write a function to search a given element x in arr[].
- A simple approach is to do **linear search.**The time complexity of above algorithm is O(n). Another approach to perform the same task is using Binary Search.
- **Binary Search:** Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Binary Search



- Video Demo:
  https://www.youtube.com/watch?time_continue=50&v=T2sFYY-fT50

Ex 02: Recursive implementation of Binary Search

```
class BinarySearch {
    // Returns index of x if it is present in arr[l..r], else return -1
    int binarySearch(int arr[], int l, int r, int x) {
        if (r >= l) {
            int mid = l + (r - l) / 2;
            // If the element is present at the middle
            if (arr[mid] == x)
                return mid;
            // If element is smaller than mid, then
            // it can only be present in left subarray
            if (arr[mid] > x)
                return binarySearch(arr, l, mid - 1, x);
            // Else the element can only be present
            // in right subarray
            return binarySearch(arr, mid + 1, r, x);
        }
        // We reach here when element is not present in array
        return -1;
    }
    public static void main(String args[]) {
        BinarySearch ob = new BinarySearch();
        int arr[] = { 2, 3, 4, 10, 40 };
        int n = arr.length;
        int x = 10;
        int result = ob.binarySearch(arr, 0, n - 1, x);
        if (result == -1)
            System.out.println("Element not present");
        else
            System.out.println("Element found at index " + result);
    }
}
```

Output:

```
Element is present at index 3
```

Ex 03: Iterative implementation of Binary Search

```
class BinarySearch {
    // Returns index of x if it is present in arr[], else return -1
    int binarySearch(int arr[], int x) {
        int l = 0, r = arr.length - 1;
        while (l <= r) {
            int m = l + (r - l) / 2;
            // Check if x is present at mid
            if (arr[m] == x)
                return m;
            // If x greater, ignore left half
            if (arr[m] < x)
                l = m + 1;
            // If x is smaller, ignore right half
            else
                r = m - 1;
        }
        // if we reach here, then element was not present
        return -1;
    }
    public static void main(String args[]) {
        BinarySearch ob = new BinarySearch();
        int arr[] = { 2, 3, 4, 10, 40 };
        int n = arr.length;
        int x = 10;
        int result = ob.binarySearch(arr, x);
        if (result == -1)
            System.out.println("Element not present");
        else
            System.out.println("Element found at "
                               + "index " + result);
    }
}
```

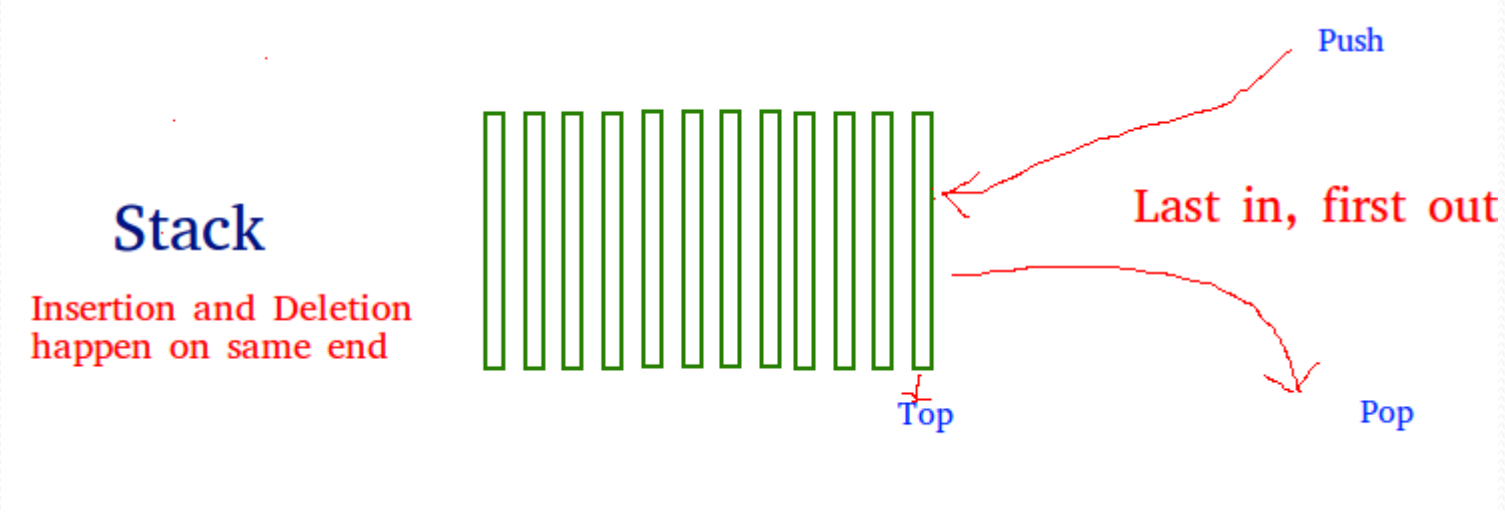
Output:

Element is present at index 3

Stacks and Queues

Stack Data Structure

- Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).



- Example : plates stacked over one another in canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time.

Operations on stacks

- Mainly the following three basic operations are performed in the stack:
- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else false.

Ex 04 : Operations on stacks

```
public class Stack {
static final int MAX = 1000;
    int top;
    int a[] = new int[MAX]; // Maximum size of Stack
    boolean isEmpty() {
        return (top < 0);
    }
    Stack() {
        top = -1;
    }
    boolean push(int x) {
        if (top >= (MAX - 1)) {
            System.out.println("Stack Overflow");
            return false;
        }
        else {
            a[++top] = x;
            System.out.println(x + " pushed into stack");
            return true;
        }
    }
    int pop() {
        if (top < 0) {
            System.out.println("Stack Underflow");
            return 0;
        }
        else {
            int x = a[top--];
            return x;
        }
    }
```

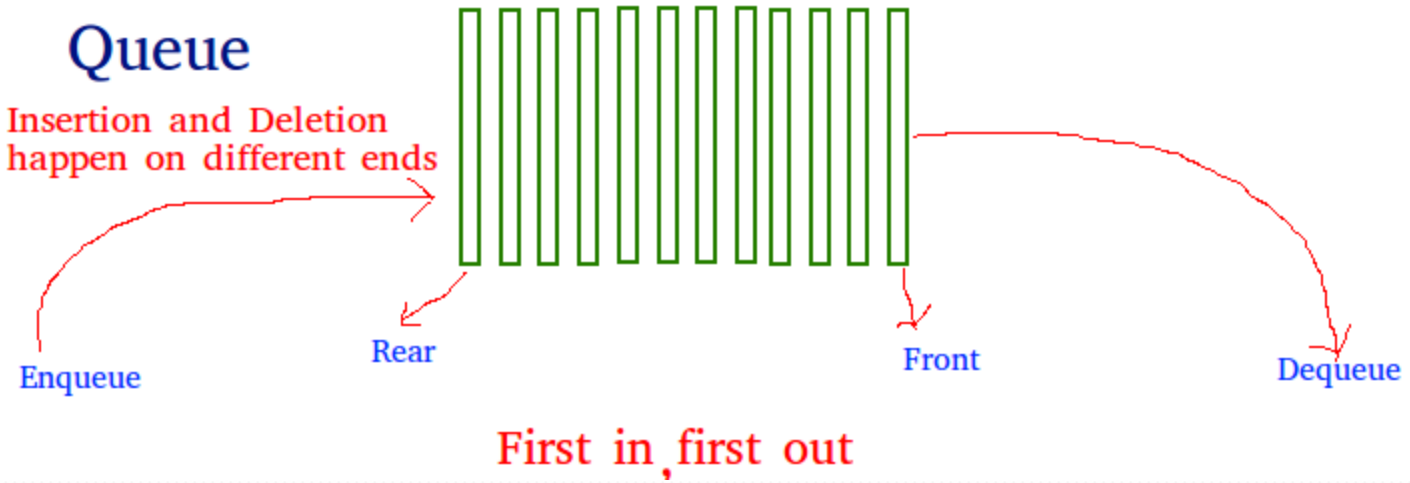
```
    int peek() {
        if (top < 0) {
            System.out.println("Stack Underflow");
            return 0;
        }
        else {
            int x = a[top];
            return x;
        }
    }

public static void main(String[] args) {
Stack s = new Stack();
        s.push(10);
        s.push(20);
        s.push(30);
        System.out.println(s.pop() + " Popped from
stack");
}
}
```

Output:

```
10 pushed into stack
20 pushed into stack
30 pushed into stack
30 Popped from stack
```

Queue Data Structure

- A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.



Array implementation of queue

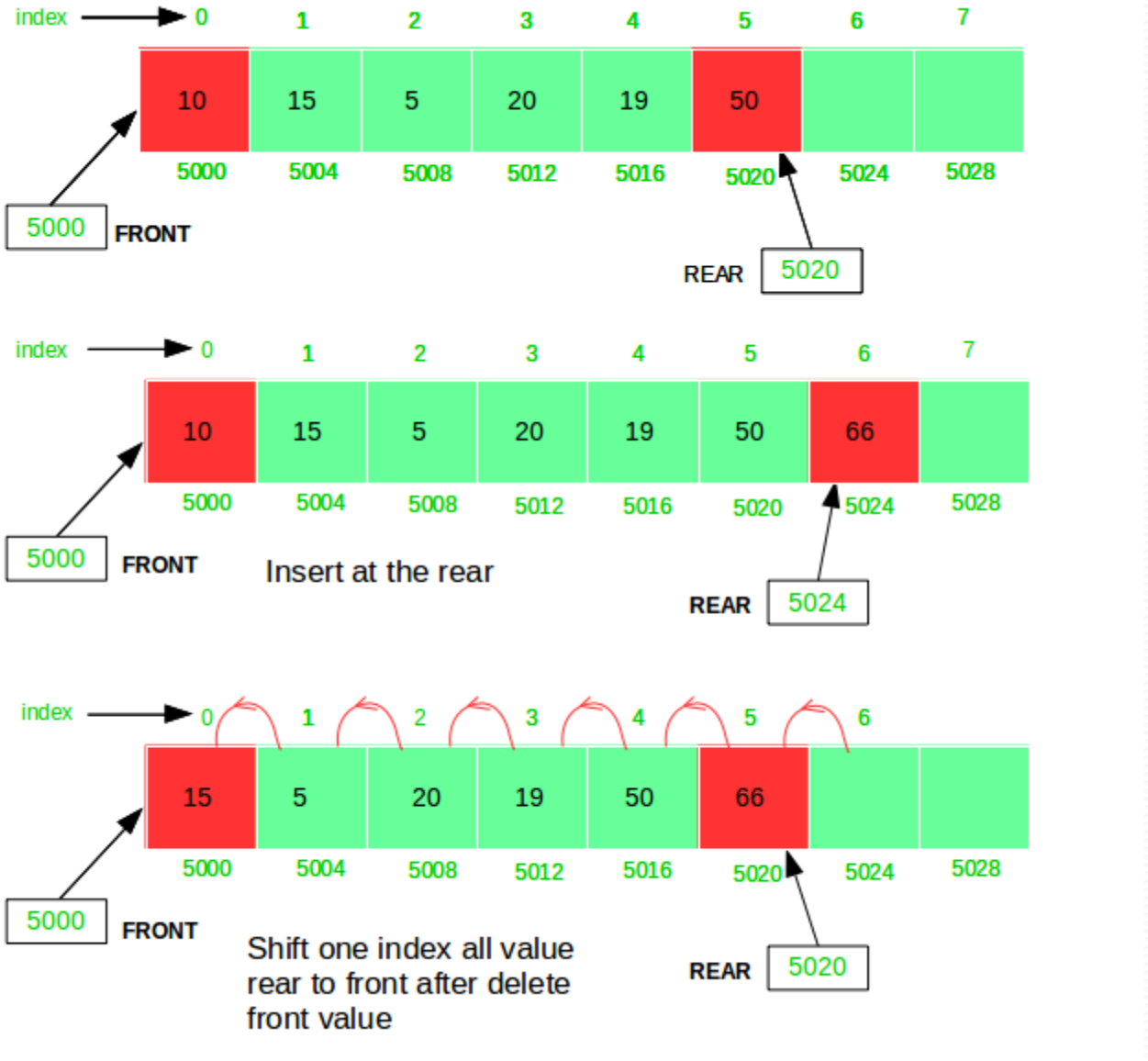
- In queue, insertion and deletion happen at the opposite ends, so implementation is not as simple as stack.

- To implement a queue using array, create an array *arr* of size *n* and take two variables *front* and *rear* both of which will be initialized to *0* which means the queue is currently empty. Element *rear* is the index upto which the elements are stored in the array (last element) and *front* is the index of the first element of the array. Now, some of the implementation of queue operations are as follows:

Queue Operations

- **Enqueue:** Addition of an element to the queue. Adding an element will be performed after checking whether the queue is full or not. If *rear* < *n* which indicates that the array is not full then store the element at *arr[rear]* and increment *rear* by *1* but if *rear* == *n* then it is said to be an Overflow condition as the array is full.
- **Dequeue:** Removal of an element from the queue. An element can only be deleted when there is at least an element to delete i.e. *rear* > 0. Now, element at *arr[front]* can be deleted but all the remaining elements have to shifted to the left by one position in order for the dequeue operation to delete the second element from the left on another dequeue operation.
- **Front:** Get the front element from the queue i.e. *arr[front]* if queue is not empty.
- **Display:** Print all element of the queue. If the queue is non-empty, traverse and print all the elements from index *front* to *rear*.

Queue


index
0
1
2
3
4
5
6
7
10
15
5
20
19
50
5000
5004
5008
5012
5016
5020
5024
5028
5000
FRONT
REAR
5020
index
0
1
2
3
4
5
6
7
10
15
5
20
19
50
66
5000
5004
5008
5012
5016
5020
5024
5028
5000
FRONT
Insert at the rear
REAR
5024
index
0
1
2
3
4
5
6
15
5
20
19
50
66
5000
5004
5008
5012
5016
5020
5024
5028
5000
FRONT
Shift one index all value rear to front after delete front value
REAR
5020

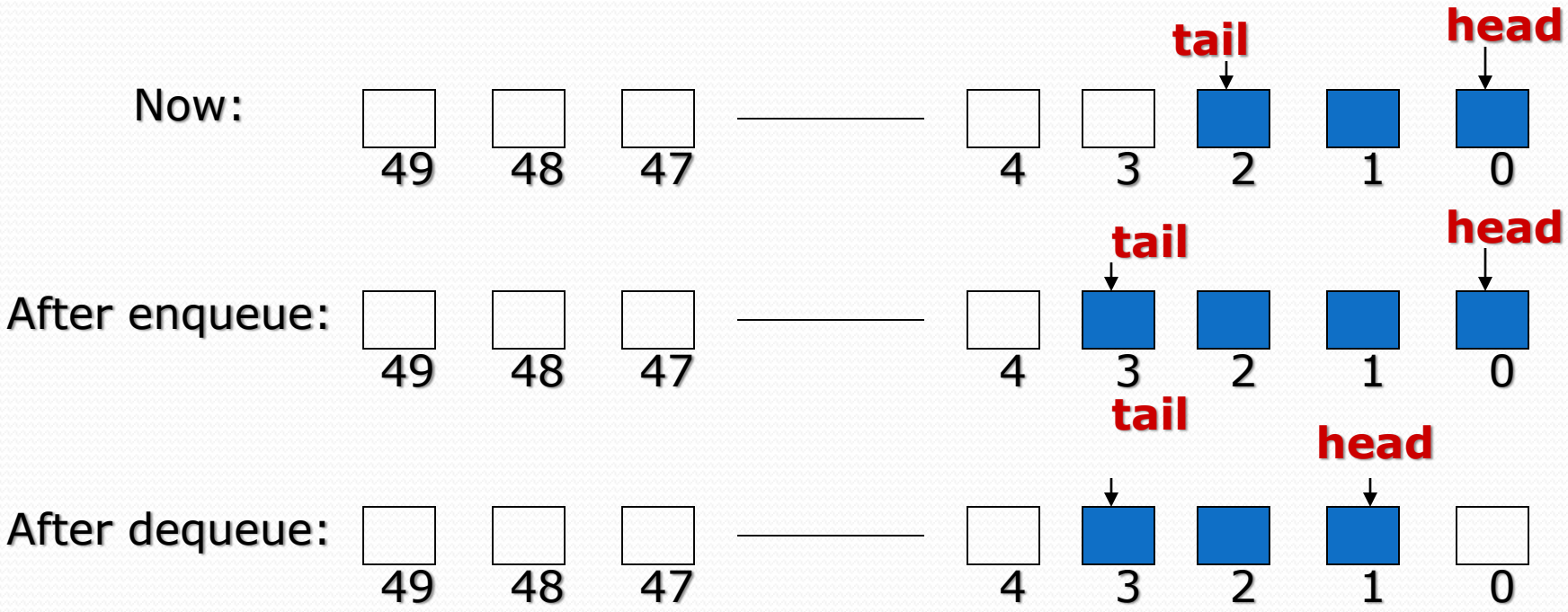
Queue Source Code

- Java code will be provided you in the lab to implement and experiment with.
- Ex 05 : StaticQueueinjava.java

Problem with simple Queue

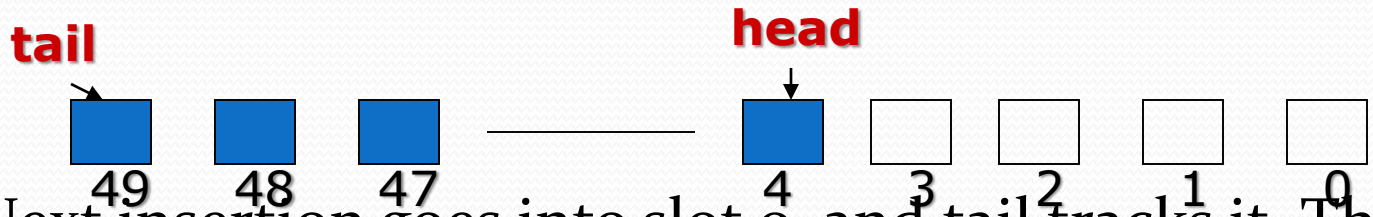
How head and tail Change

- **head** increases by 1 after each dequeue( )
- **tail** increases by 1 after each enqueue( )
- We can not fully utilize the space



Solution: A Circular Queue

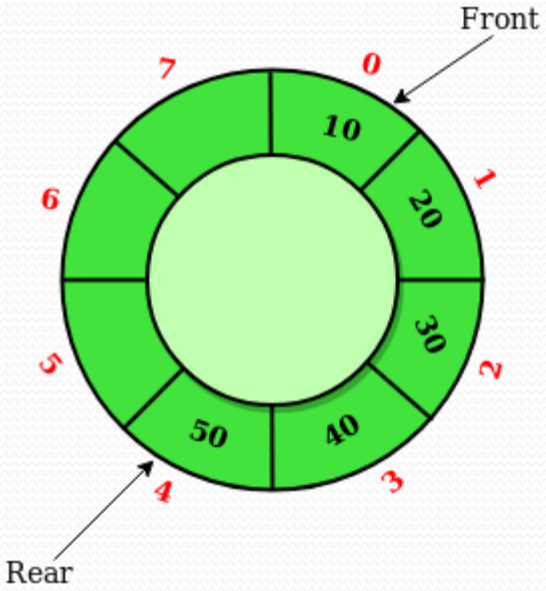
- Allow the head (and the tail) to be moving targets
- When the tail end fills up and front part of the array has empty slots, new insertions should go into the front end

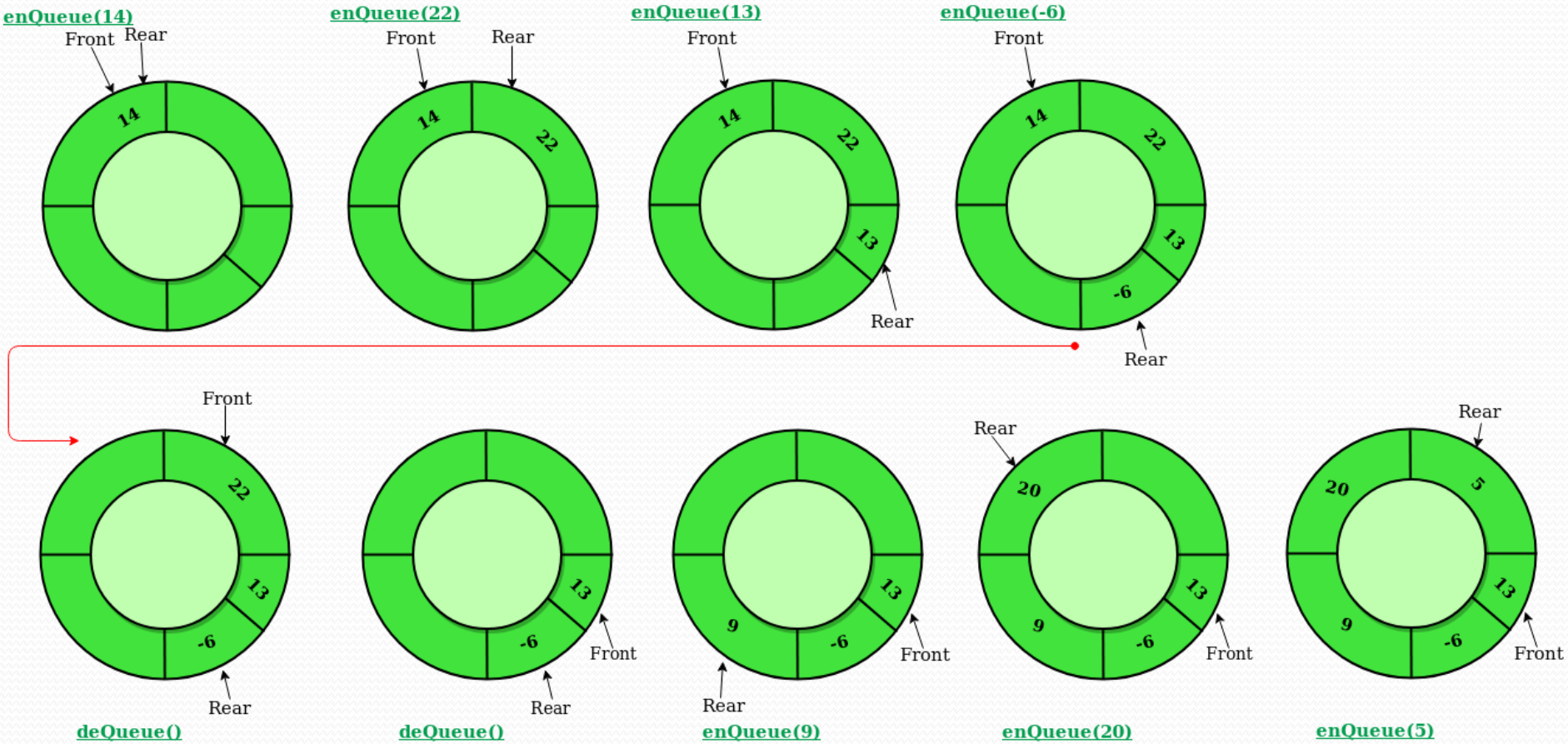


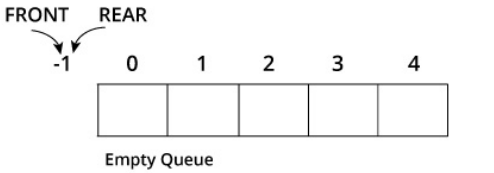
- Next insertion goes into slot 0, and tail tracks it. The insertion after that goes into a lot 1, etc.

Circular Queue

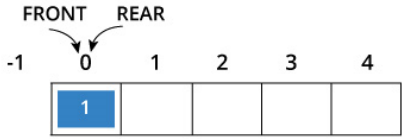
- Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called '**Ring Buffer**'.



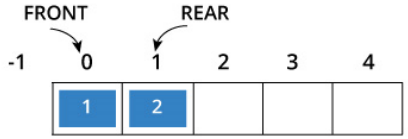

enQueue(14)
Front
Rear
14
enQueue(22)
Front
Rear
14
22
enQueue(13)
Front
14
22
13
Rear
enQueue(-6)
Front
14
22
13
-6
Rear
Front
22
13
-6
Rear
deQueue()
13
-6
Front
Rear
deQueue()
Rear
9
13
-6
Front
Rear
enQueue(9)
Rear
20
9
13
-6
Front
enQueue(20)
Rear
20
5
9
13
-6
Front
enQueue(5)



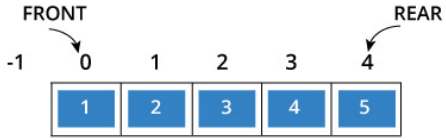
Empty Queue



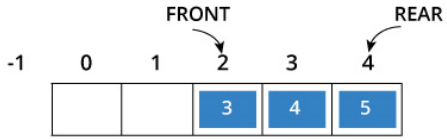
EnQueue first element



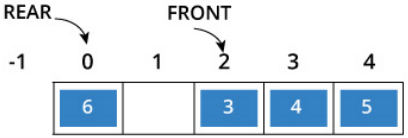
EnQueue



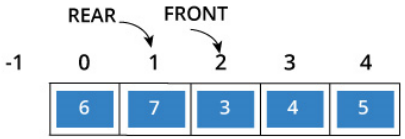
EnQueue



DeQueue



EnQueue



Queue Full

Operations on Circular Queue

- **Front:** Get the front item from queue.
- **Rear:** Get the last item from queue.
- **enQueue(value)** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.
  - **Steps:** Check whether queue is Full – Check ((rear == SIZE-1 && front == 0) || (rear == front-1)).
  - If it is full then display Queue is full. If queue is not full then, check if (rear == SIZE – 1 && front != 0) if it is true then set rear=0 and insert element.

Operations on Circular Queue

- **Front:** Get the front item from queue.
- **Rear:** Get the last item from queue.
- **enQueue(value)** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.
  - **Steps:**Check whether queue is Full – Check ((rear == SIZE-1 && front == 0) || (rear == front-1)).
  - If it is full then display Queue is full. If queue is not full then, check if (rear == SIZE – 1 && front != 0) if it is true then set rear=0 and insert element.

- **deQueue()** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position.
  - **Steps:**Check whether queue is Empty means check (front==-1).
  - If it is empty then display Queue is empty. If queue is not empty then step 3
  - Check if (front==rear) if it is true then set front=rear= -1 else check if (front==size-1), if it is true then set front=0 and return the element.

Circular Queue Source Code

- C++ code will be provided you in the lab to implement and experiment with.
- Ex 06 : MSAPW09Ex06.cpp

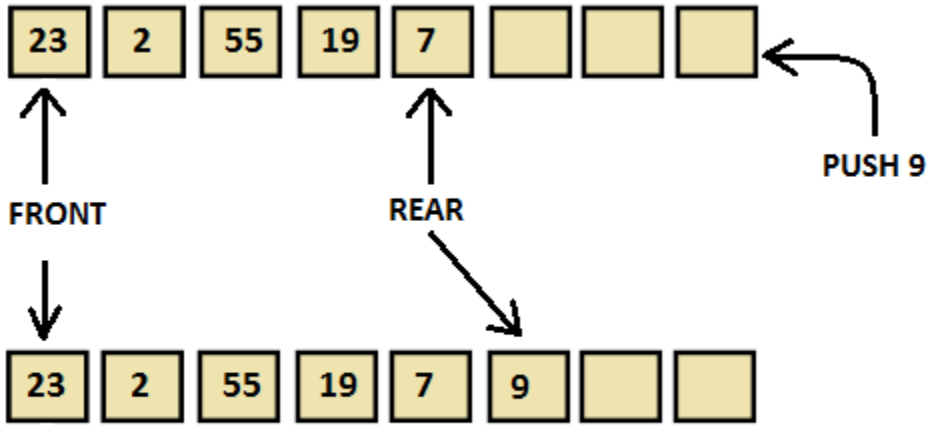
Deque or Double Ended Queue

- Deque or Double Ended Queue is a generalized version of Queue data structure that allows insert and delete at both ends.

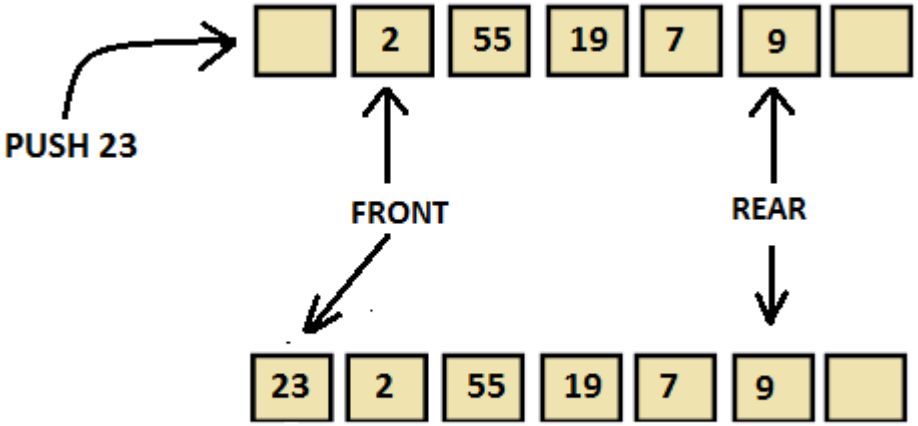
- **Operations on Deque:**
  Mainly the following four basic operations are performed on queue:
- ***insertFront()***: Adds an item at the front of Deque.
  insertLast(): Adds an item at the rear of Deque.
  deleteFront(): Deletes an item from front of Deque.
  deleteLast(): Deletes an item from rear of Deque.

Deque or Double Ended Queue

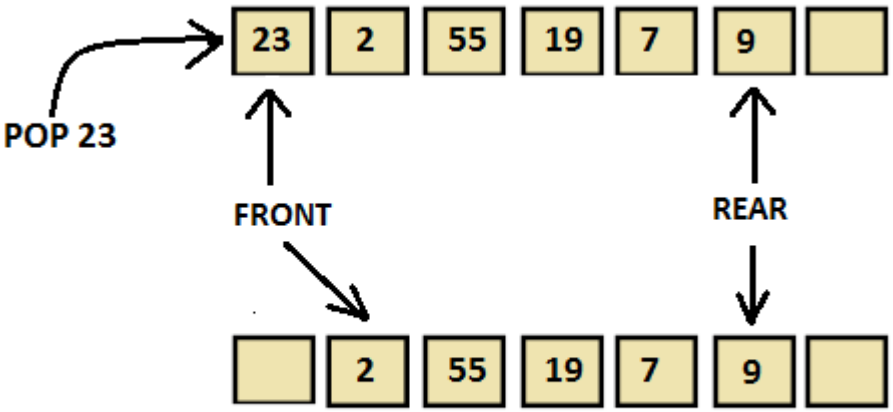
- In addition to above operations, following operations are also supported
  getFront(): Gets the front item from queue.
  getRear(): Gets the last item from queue.
  isEmpty(): Checks whether Deque is empty or not.
  isFull(): Checks whether Deque is full or not.



INSERT BACK



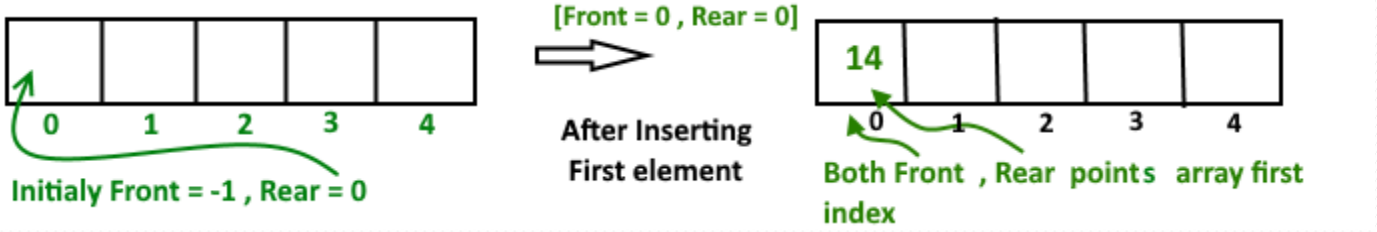
INSERT FRONT



REMOVE FRONT

Implementation of Deque using circular array

- **Circular array implementation deque**
  For implementing deque, we need to keep track of two indices, front and rear. We enqueue(push) an item at the rear or the front end of qedue and dequeue(pop) an item from both rear and front end.
  Working
  1. Create an empty array 'arr' of size 'n'
  initialize **front = -1 , rear = 0**
  Inserting First element in deque, at either front or rear will lead to the same result.



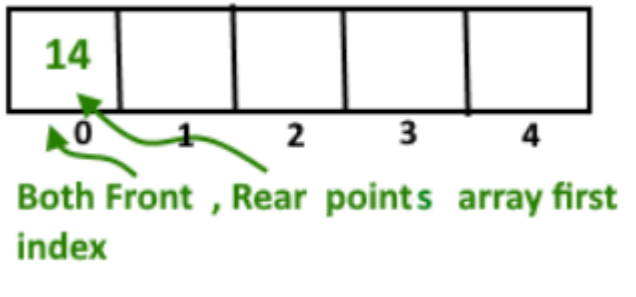
- After insert **Front** Points = 0 and **Rear** points = 0

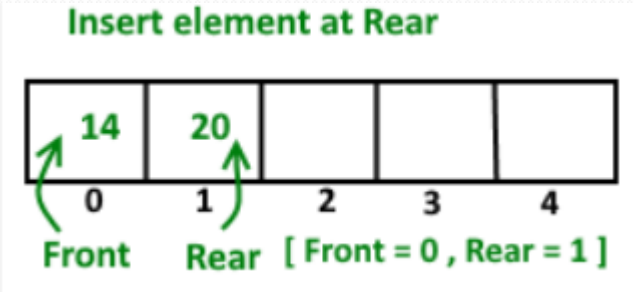
Insert Elements at Rear end

- a). First we check deque if Full or Not
- b). IF Rear == Size-1
- then reinitialize Rear = 0 ;
- Else

  increment Rear by '1' and push current key into

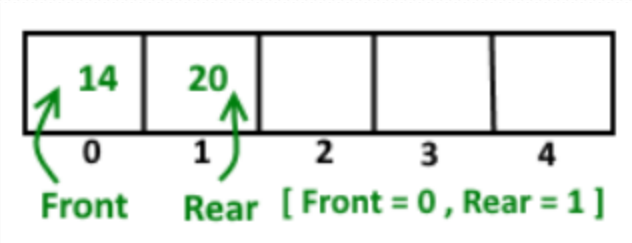
  Arr[rear] = key
- Front remain same.

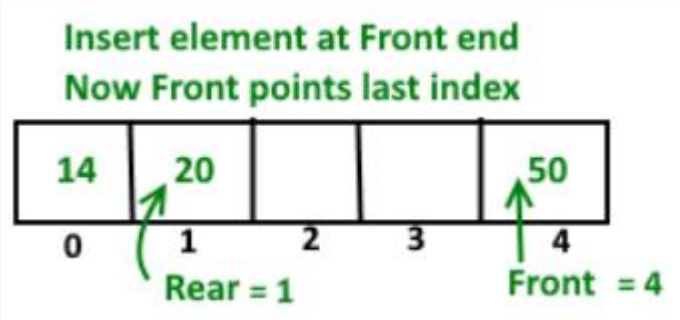




Insert Elements at Front end

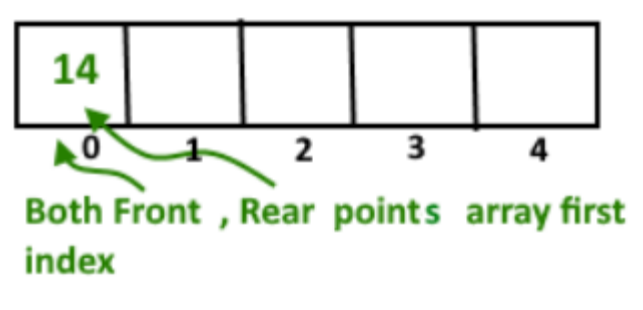
- a). First we check deque if Full or Not
- b). IF Front == 0 // initial position, move Front
- to points last index of array
- front = size - 1
- Else decremented front by '1' and push
- current key into Arr[Front] = key
- Rear remain same.

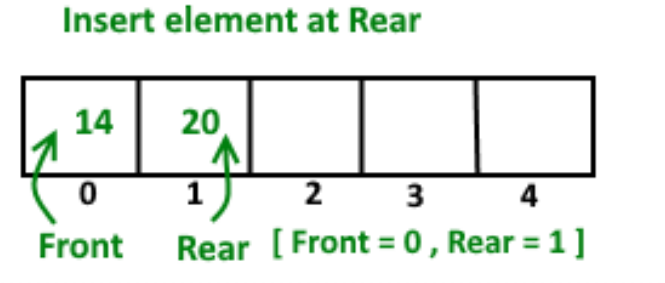




Delete Element From Rear end

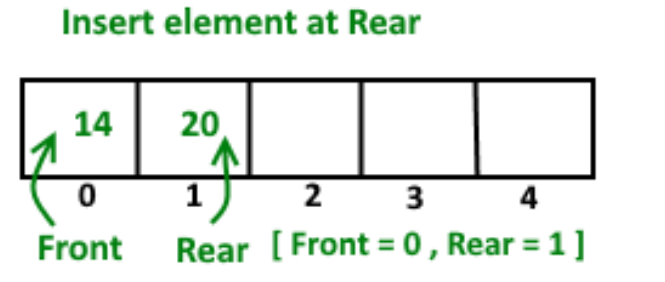
a). first Check deque is Empty or Not
b). If deque has only one element
front = -1 ; rear =-1 ;
Else IF Rear points to the first index of array
it's means we have to move rear to points last index [now first inserted element at front end become rear end]
rear = size-1 ;
Else || decrease rear by '1'
rear = rear-1;

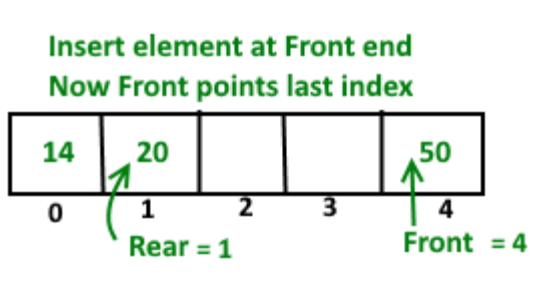




Delete Element From Front end

a). first Check deque is Empty or Not
b). If deque has only one element
front = -1 ; rear =-1 ;
Else IF front points to the last index of the array
it's means we have no more elements in array so
we move front to points first index of array
front = 0 ;
Else || increment Front by '1'
front = front+1;





Circular Queue Source Code

- Java code will be provided you in the lab to implement and experiment with.
- Ex 06 : Deque.java

Assignment #2

- Write a brief note on priority queues.
- Implement priority queues in both Java and C++.
- Deadline : Next Week.
- No print outs needed, submissions are only on slate.
- If plagiarism detected, you will get zero marks.