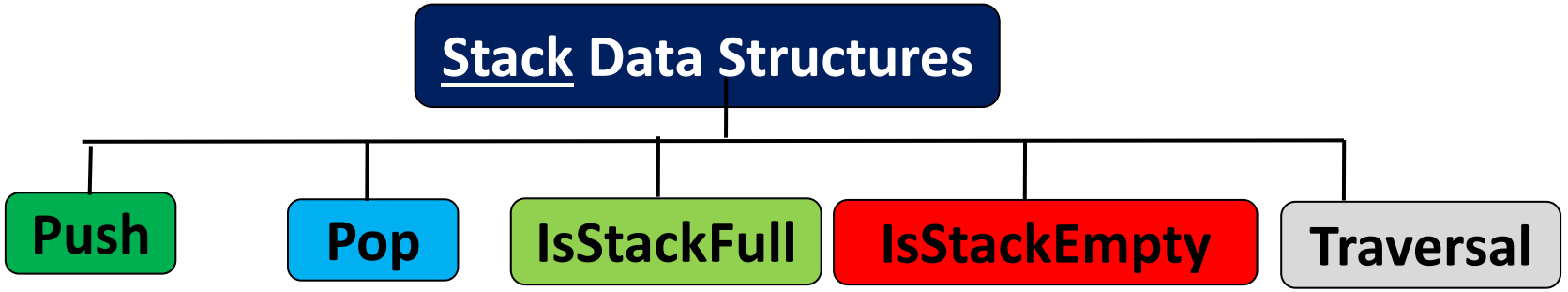
CS2x1:Data Structures and Algorithms

Koteswararao Kondepu

k.kondepu@iitdh.ac.in

Recap

- Stack (LIFO) Implementation
  - Push ()
  - Pop ()
  - IsStackFull ()
  - IsStackEmpty ()
  - PrintStack ()



- Limitations: (i) The maximum size of the stack must be defined in prior and cannot be changed; (ii) Trying to Push a new element into stack and Pop an element from the stack required an *implementation-specific exceptions*.
- Stack Application: Infix to Postfix evaluation

Stack: Exercise

Consider the following *pseudocode* that uses a stack

```
declare a stack of characters
  while ( there are more characters in the word to read ) {
     read a character
     push the character on the stack
  }
  while ( the stack is not empty )   {
     pop a character off the stack
     write the character to the screen
  }
```

What is the output for input “a*sdhdtii*"?

Stack Application: Exercise (1)

Infix to Postfix expression

Infix to Postfix expression:

Example#1: (a+-b*c)

1) a+bc*- 2) a-bc*+ 3) Invalid expression

Stack Application: Exercise (2)

Infix to Postfix expression

Infix to Postfix expression:

Example#2: a*(b+c-d/e^f*g

1) ab*c-def^/*+ 2) ab*c-def^/+* 3) Unequal Parenthesis

Infix to Postfix expression conversion Algorithm

```
top = -1, postfix[];
while (top>-1)
        token = infix.ReadingToken();
        if (token == operand) then
                append to postfix output
        elseif (token == "(" ) then
                push (token)
        elseif (token == ")" ) then
                while (token != "(" )
                        pop()
                        append to the postfix
```

// If scanned token is operand→ append to postfix output

// If scanned token is open parenthesis → push to stack

// If scanned token is closed parenthesis → pop all the operators from the stack till the open parenthesis appears

Infix to Postfix expression conversion Algorithm (1)

```
elseif (token == “operator” ) then
        if (IEP(token) > ISP(top))
            push (token)
        else
            while(ISP(top) ≥ IEP (token)
                pop()
                append to the postfix
            push(token)
```

// If In-Stack Precedence (ISP) is lower than scanned In-Expression Precedence (IEP) → then push to stack

// If In-Stack Precedence (ISP) is higher than scanned In-Expression Precedence (IEP) → then append to postfix output

Infix to Postfix expression conversion Algorithm (2)

```
elseif(token == “\0” )
        while (top != -1)
                pop()
                append to the postfix
else “Invalid expression”
```

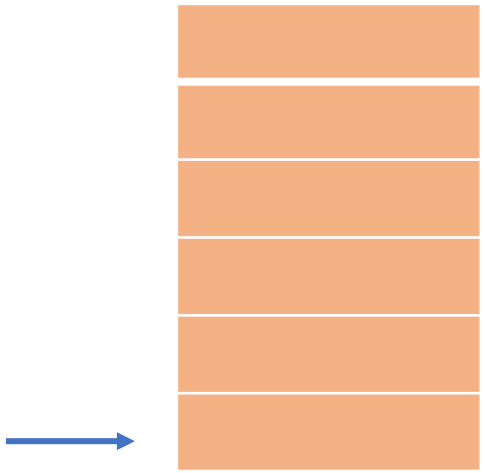
// The scanned token reached to end-of-line → pop all the remaining operators from the stack and append to postfix output

Stack Application: Exercise (3)

Infix to Postfix expression

Infix to Postfix expression:

Example#3: a+b*c/(d-e)^g^h



is pushed into stack
is pushed into stack
is popped from stack
is pushed into stack
is pushed into stack
is pushed into stack
is popped from stack
is pushed into stack
is popped from stack
is pushed into stack
is popped from stack
is popped from stack
is popped from stack

Assignment#1

- Objective: Implement the infix expression to postfix expression conversion *using stack operations*.
- Inputs: A file (e.g., input.txt) as a command-line argument
  - What the input file should contain?
    - The input file contains the arithmetic expression

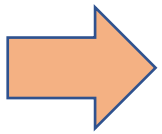
- Output: A file (e.g., output.txt)
  - What the output file should contain?
    - Should output error message if **input expression is *invalid expression***

    - If the scanned expression is a **valid infix expression** then your program has to convert it to a **valid postfix expression and print each stack operation** and the corresponding postfix expression in the *output.txt*.

Assignment#1: Example

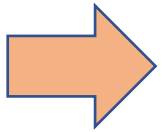
- Objective: Implement the infix expression to postfix expression conversion *using stack operations*.
- Inputs: A file (e.g., input.txt) as a command-line argument
  - What the input file should contain?
    - The input file arithmetic expression
- Output: A file (e.g., output.txt)
  - What the output file should contain?
    - Should write a message if **input expression** is **"Invalid expression"**
    - Should write a message if **input expression** is "**Unequal Parenthesis**"
    - If the scanned expression is a **valid infix expression** then your program has to convert it to a **valid postfix expression and print each stack operation** and the corresponding postfix expression in the *output.txt*.

input.txt



*a+b*c/(d-e)^g^h*

output.txt



+ is pushed into stack
* is pushed into stack
* is pushed into stack
/ is pushed into stack
(is pushed into stack
- is pushed into stack
- is popped from stack
(is popped from stack
^ is pushed into stack
^ is popped from stack
^ is pushed into stack
^ is popped from stack
/ is popped from stack
+ is popped from stack
Postfix expression:
abc*de-g^h^/+

Abstract description

- Step 1: Check if your program has correct number of command line argument!
- Step 2: Open the file in read mode and provide an exception in case if file not found.
- Step 3: Check whether the infix expression is a valid or not
- Step 4: If the infix expression valid then

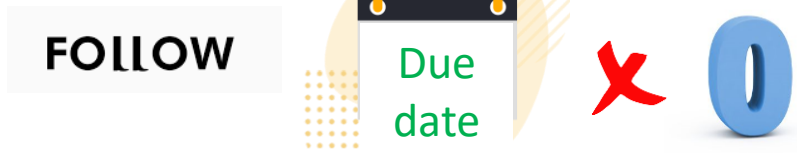
scan (each token of the file) *//while, for, do-while*
begin
evaluate the infix to postfix conversion algorithm
end

obtain the stack operations and postfix expression

- Step 5: write the stack operations and postfix expression into *output.txt* file as requested in the assignment.
- Step 6: Submit the source file as requested within the *due date*.

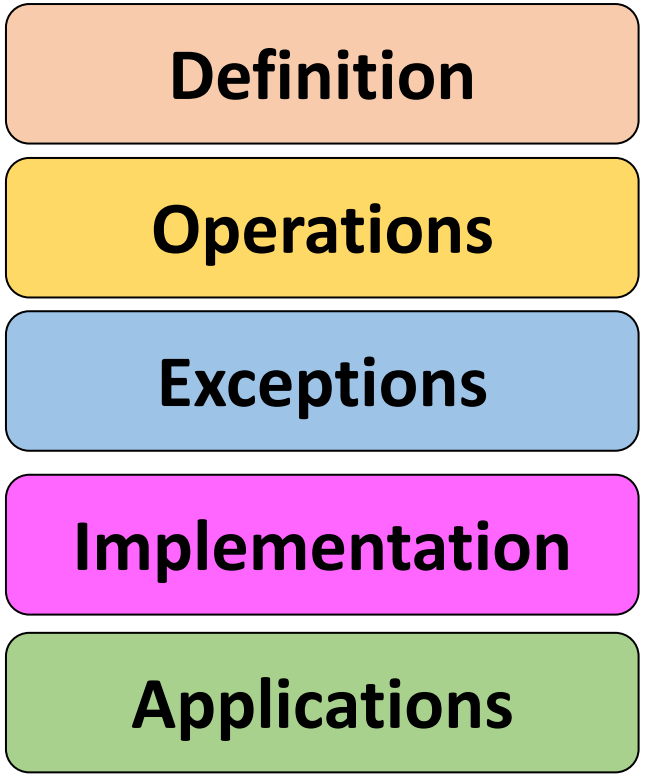
Submission and Evaluation

- The program YOU submit should output: "**output.txt**" when we run for the evaluation
- The main file of your program should be named as **<roll no>.<extension>** Ex: **220010001.c**
- Test well before submission.
  - YOU may use the provided sample input files for testing and also corresponding output files.
  - The mark YOU obtain is purely based on whether your program correctly gives outputs for the hidden inputs provided by us.
  - If your program has only a single source file, please submit the file as it is. If your program has multiple source files, please submit your code as a zip file where the name of the zip file should be your **<roll no>**
  - **It is important that you follow the input/output conventions exactly** (including the naming scheme) as we may be doing an automated evaluation.
  - ***There will be a penalty of 10% (on the mark you deserve otherwise) if you do not follow the naming conventions exactly***.
  - Submit only through moodle . ***Submit well in advance***.



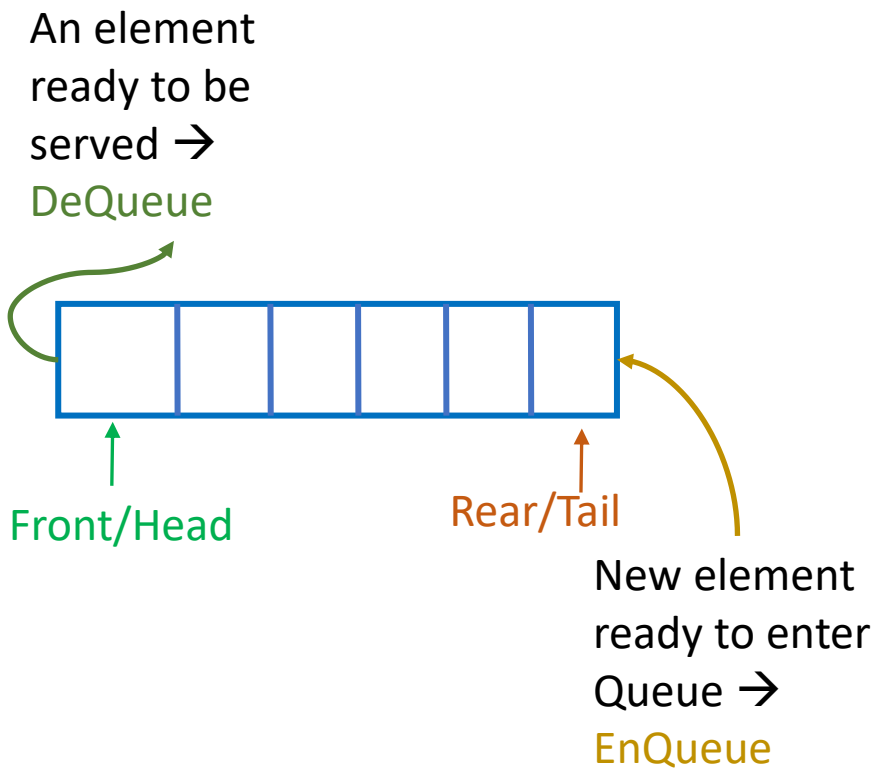
Outline

- Queue data structures
- Define Queue data structures
- Queue operations
- Queue exceptions
- Queue implementation
- Queue implementation demonstration
- Queue applications



Define: Queue

- *Queue is a two side open data structure used for storing data*
  - *First-In-First-Out (FIFO) or Last-In-Last-Out (LILO)*
  - Insertions are done at one end → Rear/Tail/Back
  - Deletions are done at another end → Front/Head



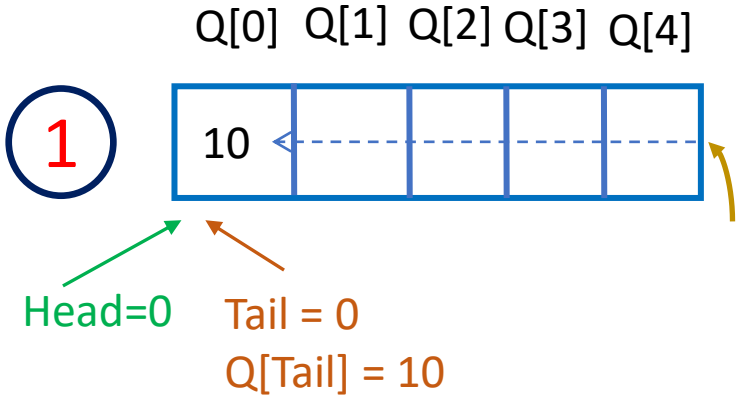
- **Major Operations**:
  - An element is inserted in a Queue → ***EnQueue***
  - An element is removed from the Queue → ***DeQueue***

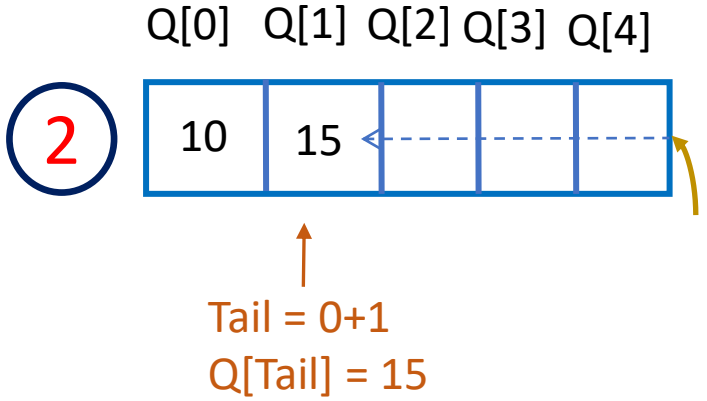
- **Exceptions**:
  - ***Underflow*** – Trying to **DeQueue** from an *empty queue*
  - ***Overflow*** – Trying to **EnQueue** an element in a *full queue*

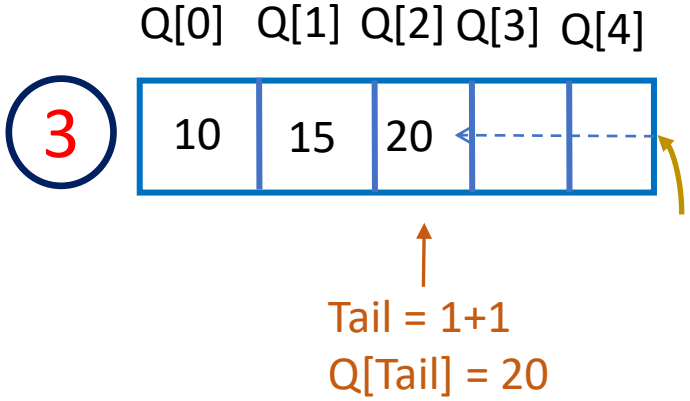
Queue: EnQueue Example

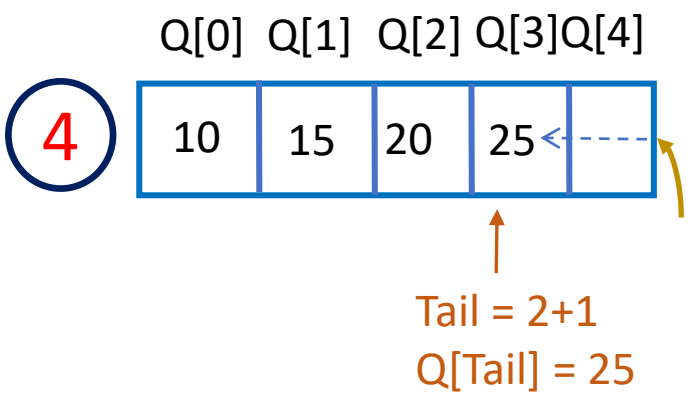
```
Initial begin
    QueueSize = 5
    Head = Tail = -1
end
```

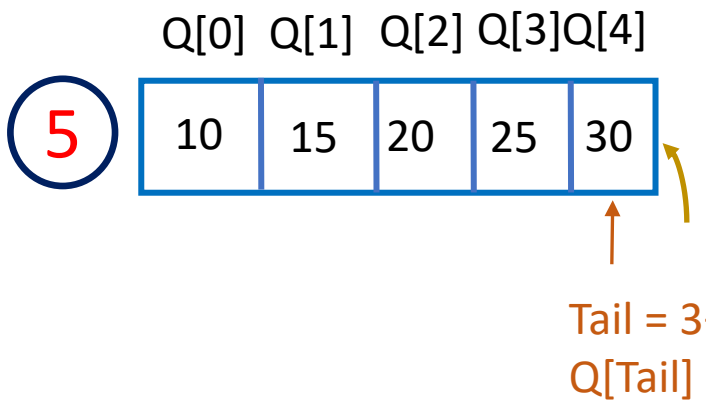
- *Queue is a two side open data structure used for storing data*
  - *First-In-First-Out (FIFO) or Last-In-Last-Out (LILO)*

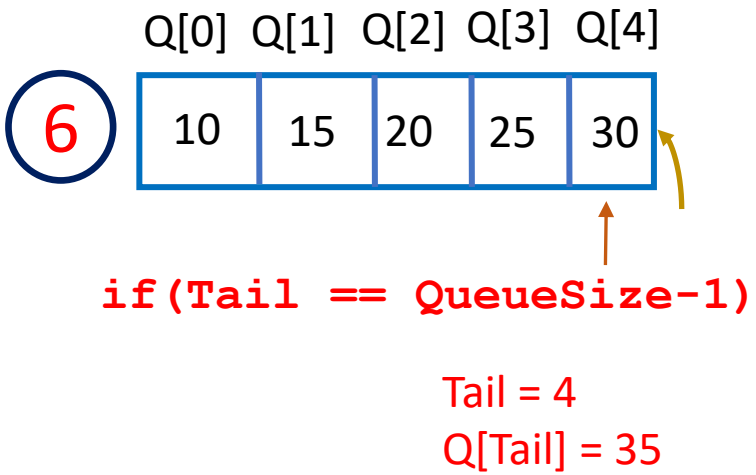




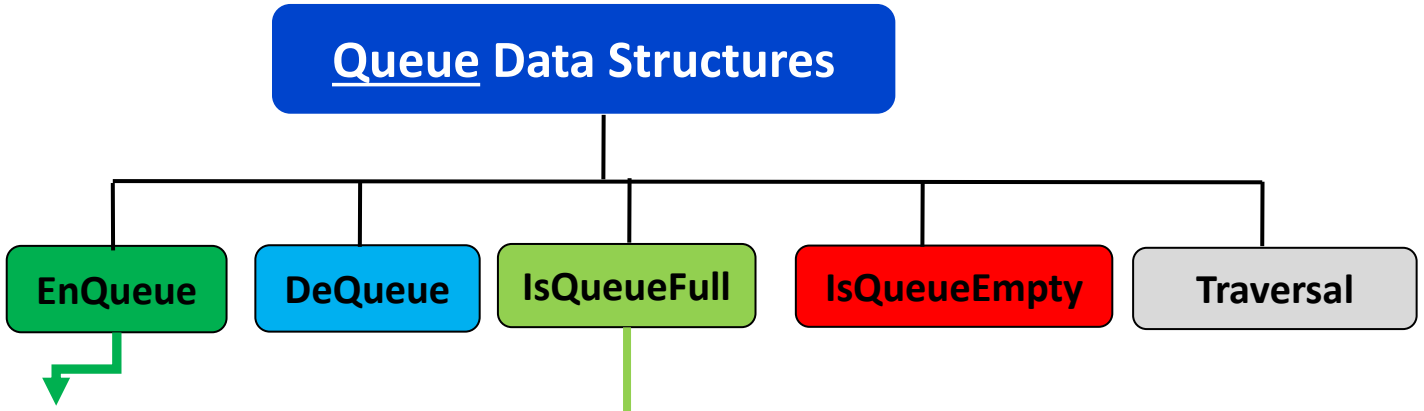








Implementation: EnQueue



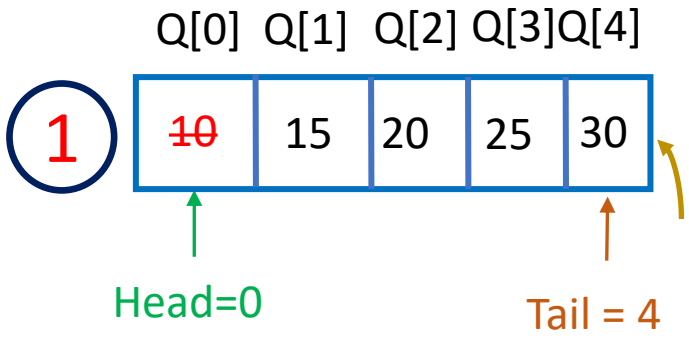
Head = Tail = -1

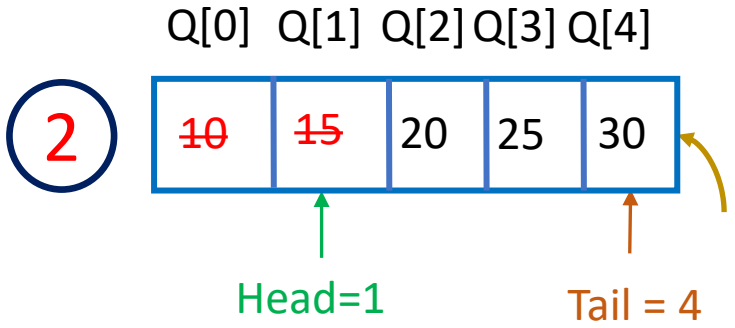
```
void EnQueue(){
  int Element;
  if(!IsQueueFull()){
    printf("Enter the element to be inserted into the Queue\n");
    scanf("%d", &Element);
    Tail++;
    Queue[Tail] = Element;
    if(Head == -1)
    Head++;
  }
  else{
    printf("Enqueue is not possible as Queue is Full (overflow)\n"); }}
```

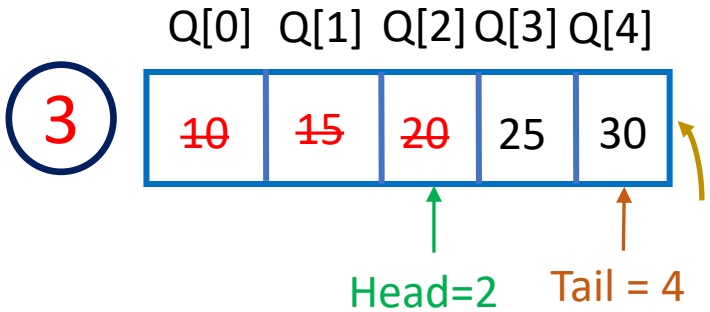
```
int IsQueueFull(){
if(Tail == QueueSize-1)
    return 1;
else return 0;}
```

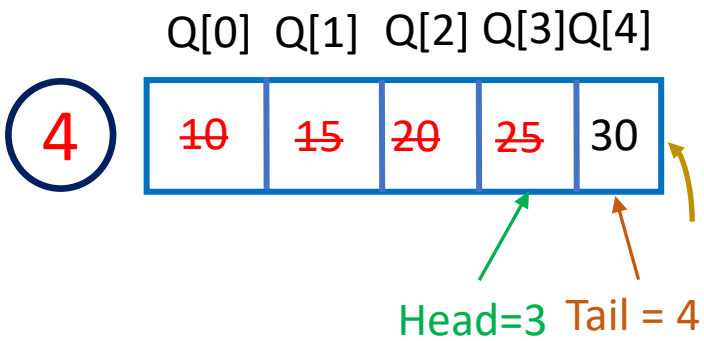
Queue: DeQueue Example

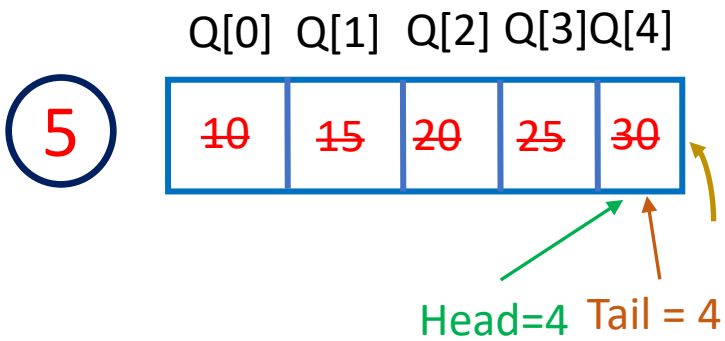
```
Initial begin
    QueueSize = 5
    Head = Tail = -1
end
```

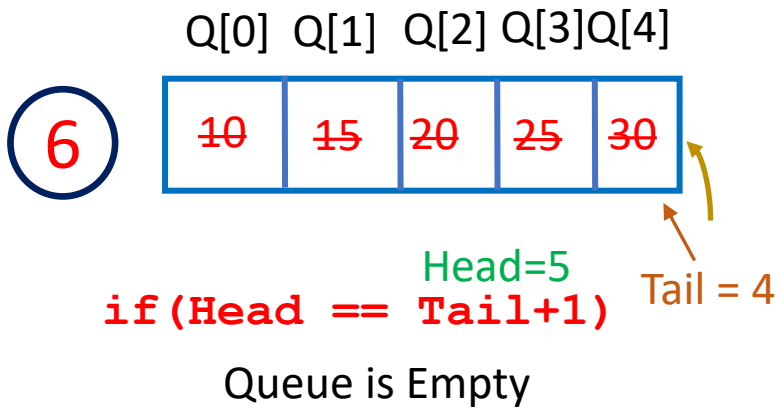




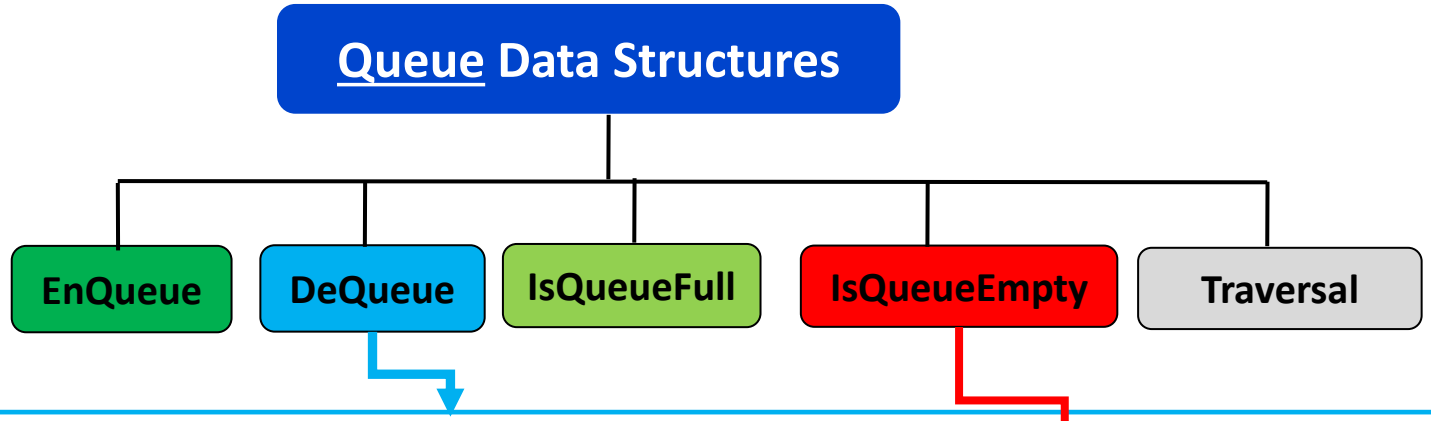








Implementation: DeQueue

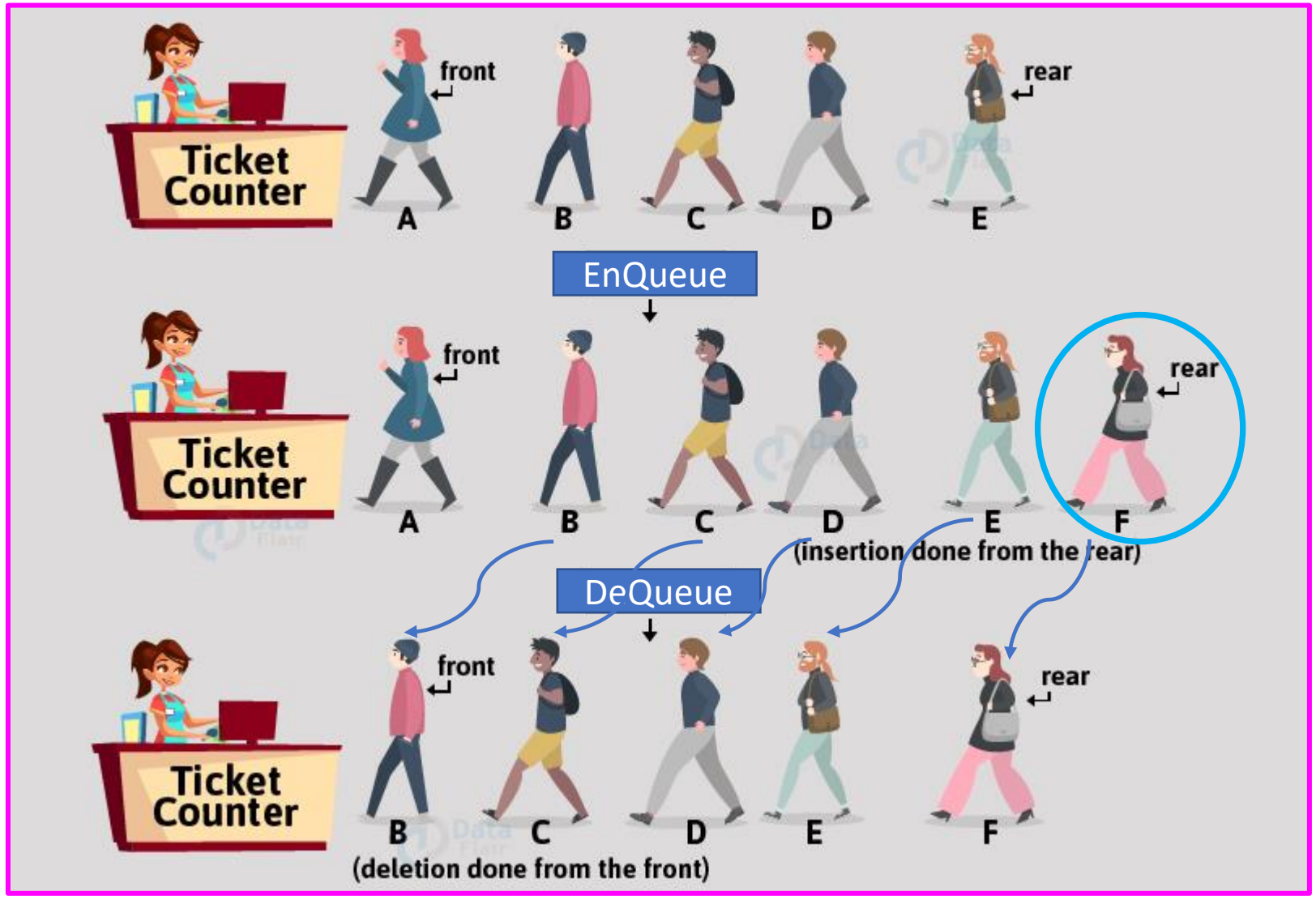


Head = Tail = -1

```
void DeQueue(){
  if(!IsQueueEmpty()){
    printf("%d is deleted from the queue\n",
    Queue[Head]);
    //Queue[Head] = -1 *****;
    Head++;
  }
  else{
    printf("DeQueue is not possible as Queue is already Empty
(Underflow \n");
  }
}
```

```
int IsQueueEmpty(){
  if(((Head == -1)&&
      (Tail == -1))||
    (Tail+1 == Head)){
    return 1;
  } else return 0;
}
```

Queue: EnQueue and DeQueue in Reality

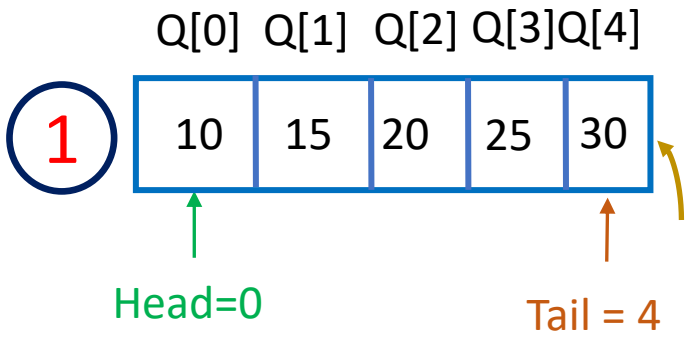


[How Queues are Implemented with Arrays & Linked List - DataFlair (data-flair.training)]

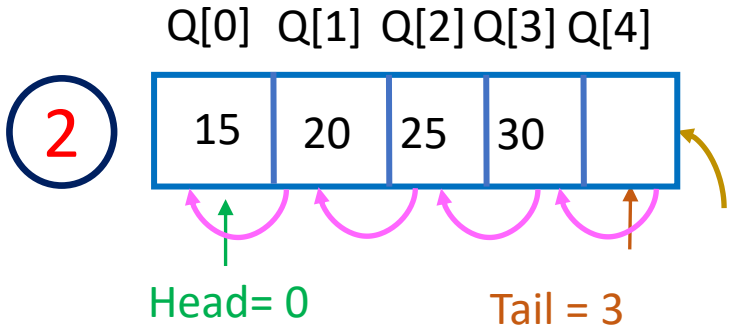
Queue: DeQueue-Real Example

```
QueueSize = 5
Head = 0
Tail = 4
```

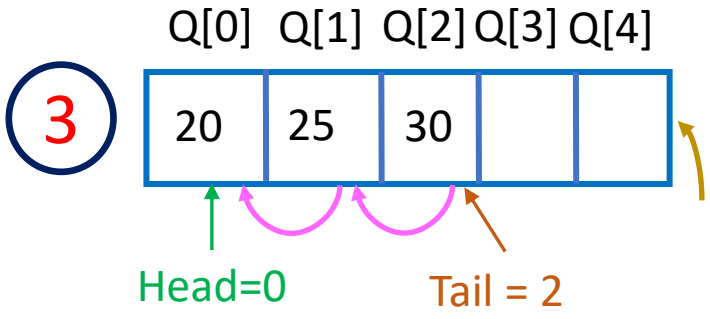
DeQueue (10)



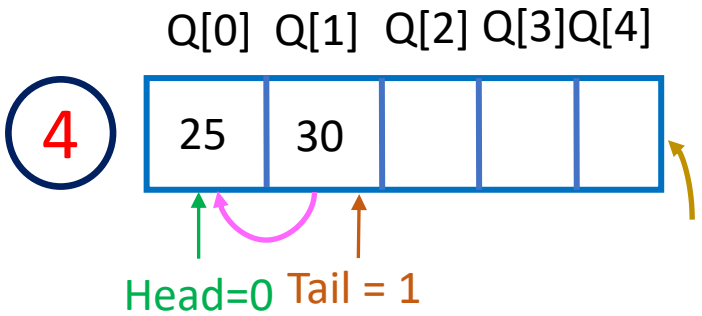
DeQueue (15)

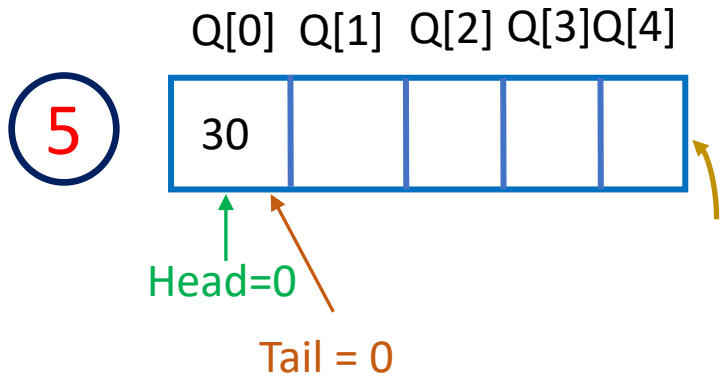


DeQueue (20)

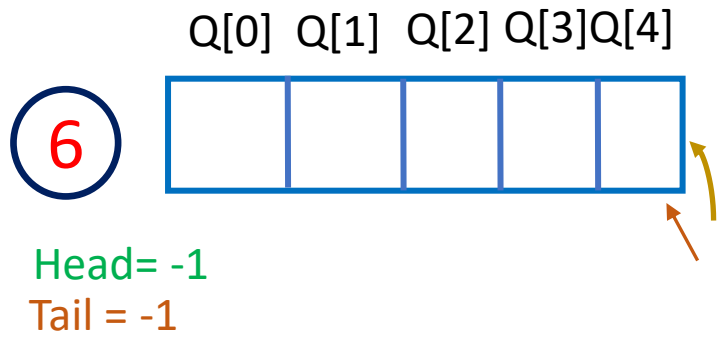


DeQueue (25)





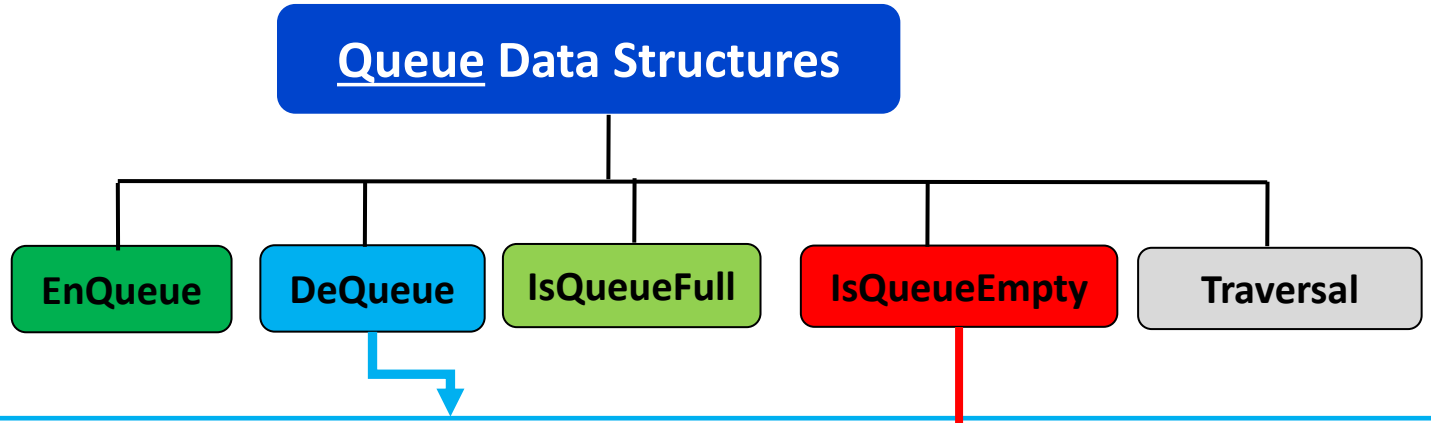
DeQueue (30)



`If((Head == -1 && Tail == -1))`

Queue is Empty

Implementation: DeQueue-Real



Head = Tail = -1

```
void DeQueue(){
 if(!IsQueueEmpty()){
    printf("%d is deleted from the queue\n", Queue[Head]);
    for (i=0; i<Tail; i++){
    Queue[i] = Queue[i+1];}
 }
   Queue[Tail] = -1*****;
    if (Tail == 0) {
       Tail = -1; Head = -1;}
    else Tail--;

else{  printf("DeQueue is not possible as Queue is already Empty (Underflow \n");
  }}
```

```
int IsQueueEmpty(){
  if(((Head == -1)&&
      (Tail == -1)){
    return 1;
  } else return 0;
}
```

Queue: Exercise

The initial configuration of a queue [size=4] is 10,20,30,40, ('10' is in the front end). To get the configuration 40,30,20,10, one needs a minimum of

[Note: The consider the Queue is a *real Queue*]

(a) 2 deletions and 3 additions
(b) 3 deletions and 2 additions
(c) 3 deletions and 3 additions
(d) 3 deletions and 4 additions

Queue: Exercise (1)

After performing the following operations on Queue data structure [assume queue size is 5], what is the output?
Enqueue(1), Enqueue(3), Enqueue(5), Enqueue(7), Dequeue(1), Enqueue(9), Dequeue(1)

(a) 3,5,7,9; (b) Error ; (c) 5, 7, 9; (d) None

thank you!

email:
k.kondepu@iitdh.ac.in

NEXT Class: 25/04/2023