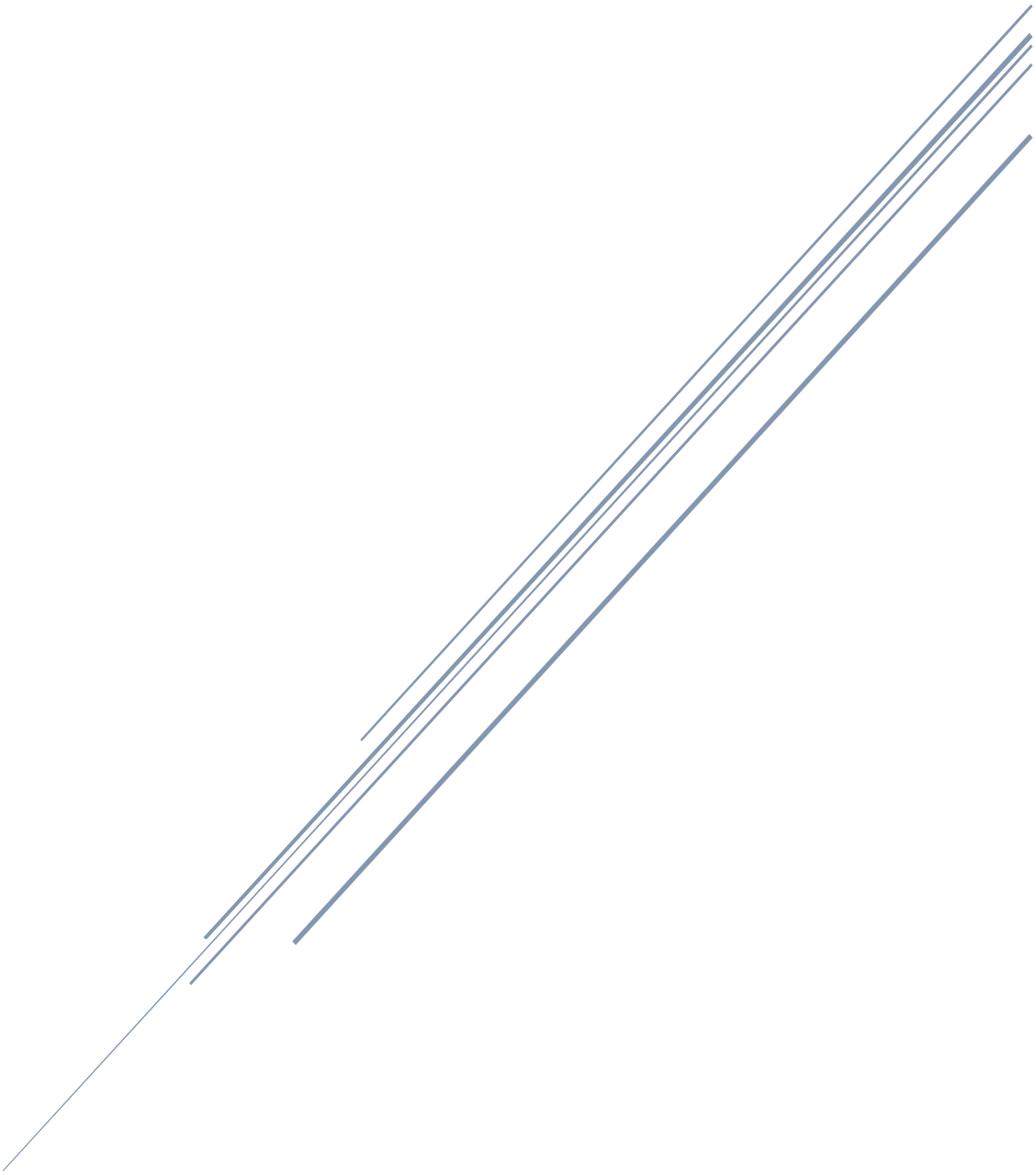
STACKS AND QUEUES

Chapter 02



Prakash Hegade
Minor Degree Program – School of CSE, KLE Tech.

Stacks

If you are a fast food lover and the title reminded you of lay's stax potato chips, then you have almost understood the concept already. If you look out for the standard dictionary meaning for the word stack, you would end up reading 'a pile of objects, typically one that is neatly arranged'. Well, that is it! You comprehend the concept already.

What remains is the mathematical perception and how a computer engineer sees it all.

Introduction

Let us consider an array with the name 'array' of size 'n'. You can access any element of the array by specifying the appropriate index. Visualize the picture mentally and this is how probably it should look like:

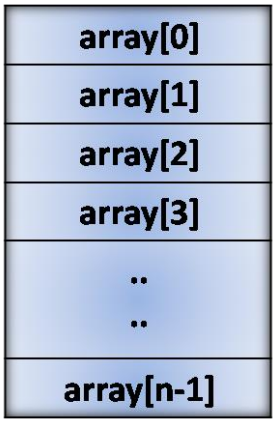


Fig: An array of size n

Let us now understand how the array becomes the stack. Look at the same array and apply the following conditions:

- Seal the three sides of the array by leaving the top or bottom end open
    - Say the top end is left open
    - To be more specific, we can only see what is at the top of the array
- Imagine an index which will allow us the access of top most element of stack
    - Say the index is named as 'top'

Visualize all the conditions on the figure above.

It would be a lot better if you draw the new diagram by applying above conditions in your rough work and move to the next page. I hope your thought process matches the figure.

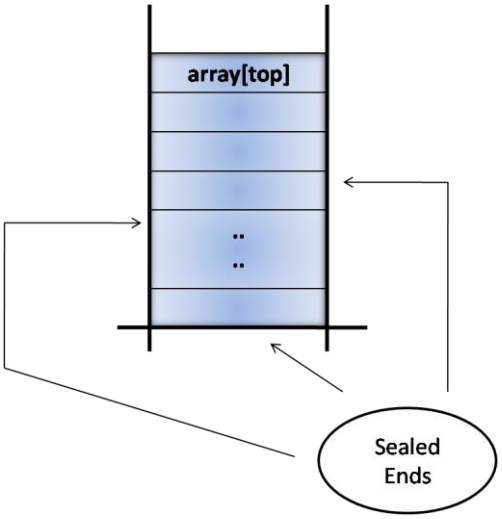


Fig: A Stack

With this understanding, let us note down the basic properties of stack:

- We can insert a new item only on top of the stack.
- We can delete an existing item only from top of the stack.
- We can only see what is at the top of the stack.

Stack represents a class of problems where the operations happen at one end. Let us try to identify some real world examples which exhibit the same behavior as stack. Following are a few:

- Parle Poppins
- Sequence of dreams in the movie Inception
- Bangles worn in hand
- Rings put inside the well
- Disks placed one above the other in disk stand

Can you think of more examples?

Definition

Before we formally study the definition, let us fix with the naming convention for the operations.

Si. No.	Name	Description of the Operation
1.	PUSH	Insert an item into the stack.
2.	POP	Delete an item from the stack.
3.	PEEK	Look out for the top most item from the stack.
4.	UNDERFLOW	Stack is empty. PEEK / POP are not allowed.
5.	OPERATIVE	PUSH / POP / PEEK operations are allowed.
6.	OVERFLOW	Stack is full. PUSH is not allowed.

Table: Stack Operations

Implementation

Given the definition, we will now do its implementation. The implementation will be done step wise slowly building up the functions one by one.

Problem

Implement a stack which can hold a maximum of 100 integer values. Use the following structure for the stack:

```
#define STACKSIZE 100
struct stack {
        int top;
        int items[STACKSIZE];
};
typedef struct stack STACK;
```

Support the implementation with the Push, Pop and Peek functions. Also write a Print function to print all the contents of the stack. The integer items, logically, have to be displayed from the top to zero indexes.

Give a user menu in the main() so that user can select the required operation to perform. Print meaningful messages after every operation.

Solution:

Let us first visualize the structure that is given to us with appropriate initializations.

Let us say for the given structure,

```
#define STACKSIZE 100
struct stack
{
   int top;
   int items[STACKSIZE];
};
typedef struct stack STACK;
```

We will create a variable of type STACK and initialize the top value.

```
STACK s;
s.top = -1
```

In memory, this would look something like following:

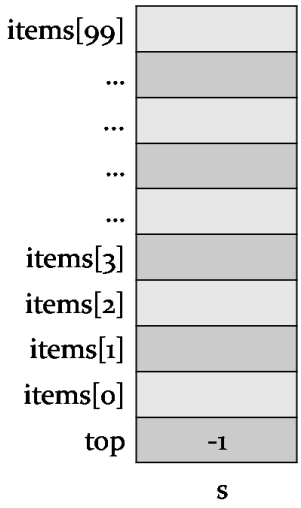


Fig: Memory layout for initialized stack

We have initialized the top to -1. Now what would be the stack memory layout for 'Underflow' condition? Can you visualize that in your mind?
The state is in underflow when there are no more items left in stack to pop. The stack is empty with 'n' number of push and 'n' number of pop.

Let us now imagine the 'Overflow' state. It is overflow when there is no more room for push. The memory layout would look as following as shown in figure below:

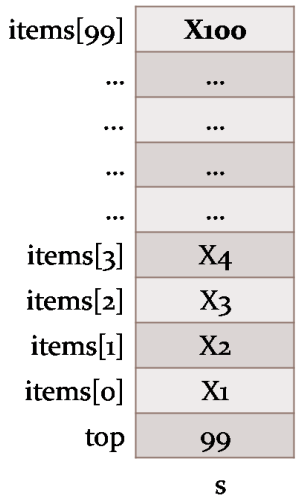


Fig: Memory layout for Overflow condition

Let us now understand the push operation. Top is initialized to -1, so first we need to increment the top and then push an item x into the stack. The steps are explained in the figure below.

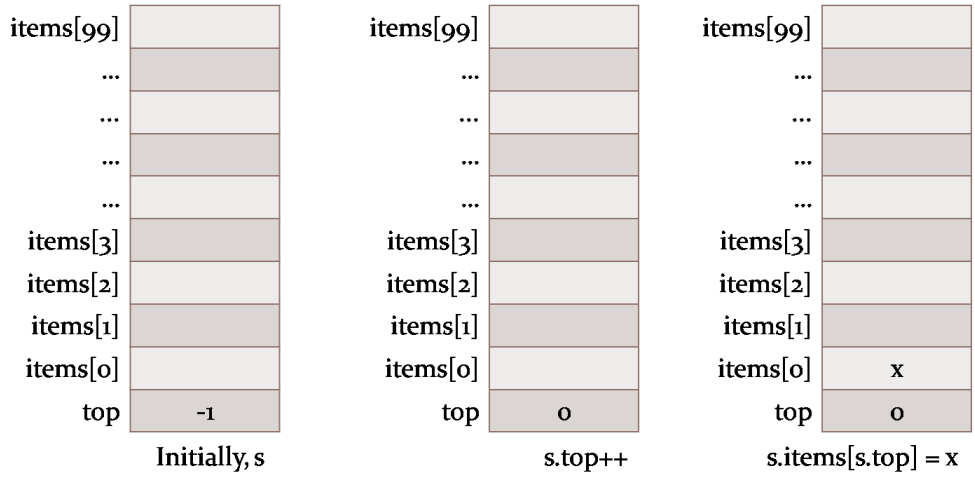


Fig: Memory layout for Push Operation

From the above figure we can note that push operation is made up of two simple steps.

PUSH

- Increment the top → s.top++
- Copy the item → s.items[s.top] = x

Similarly let us also understand the pop operation. In pop, we first copy the item and then decrement the top. Essentially, the operation is reverse of push operation. Figure below explains the steps.

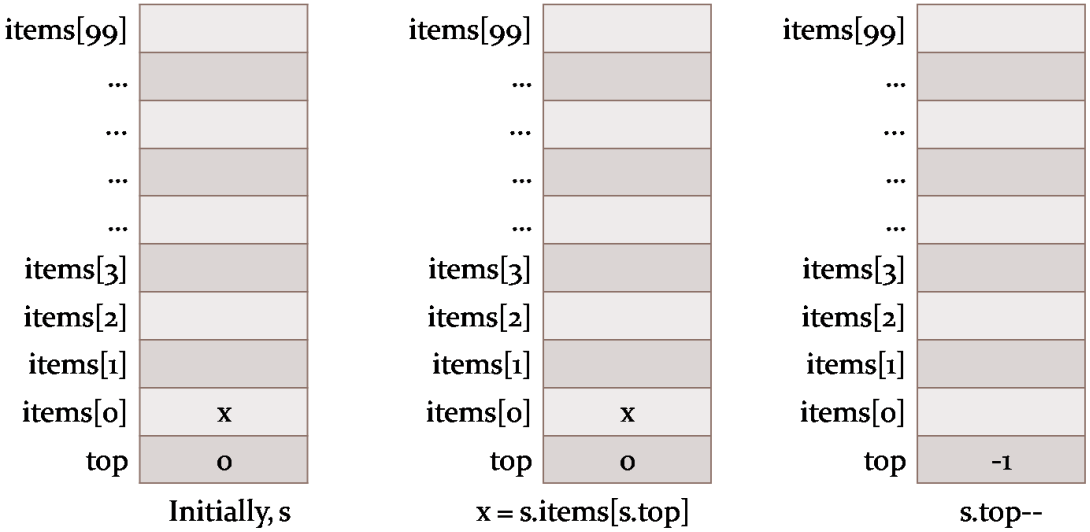


Fig: Memory layout for Pop Operation

The way we defined Push, we can define Pop operation in two steps as:

POP

- Copy the item to temporary variable x → x = s.items[s.top]

- Decrement the top → s.top--

With similar conventions, Peek operation is accessing the top most items in the stack. We now have sufficient knowledge to implement the stack.

```
#include <stdio.h>
#include <stdlib.h>
#define STACKSIZE 5
#define TRUE 1
#define FALSE 0

struct stack
{
    int top;
    int items[STACKSIZE];
};
typedef struct stack STACK;
void push(STACK *);
void pop(STACK *);
void print(STACK *);
void peek(STACK *);
int empty(STACK *);
int full(STACK *);

int main()
{
    STACK S;
    S.top = -1;
    int choice=0;
    while(1) {
        printf("\n Menu\n");
        printf("1-PUSH\n");
        printf("2-POP\n");
        printf("3-PEEK\n");
        printf("4-PRINT\n");
        printf("5-EXIT\n");
        printf("Enter your choice\n");
        scanf("%d", &choice);
        switch(choice) {
            case 1: push(&S);
                    break;
            case 2: pop(&S);
                    break;
            case 3: peek(&S);
                    break;
            case 4: print(&S);
```

```
                    break;
            case 5: printf("Terminating\n");
                    exit(1);
        }
    }
    return 0;
}

int full(STACK *S)
{
    if(S->top == STACKSIZE-1)
        return TRUE;
    else
        return FALSE;
}

void push(STACK *S)
{
    if(full(S)){
        printf("Stack full\n");
        return;
    }
    int x;
    printf("Enter the item to be pushed\n");
    scanf("%d", &x);
    S->top++;
    S->items[S->top] = x;
}

int empty(STACK * S)
{
   if(S->top == -1)
        return TRUE;
   else
        return FALSE;
}

void pop(STACK *S)
{
    if(empty(S)){
        printf("Stack Empty\n");
        return;
    }
    int x;
    x = S->items[S->top];
    printf("Popped item is %d\n", x);
```

```
    S->top--;
}

void peek(STACK *S)
{
    if(empty(S)){
        printf("Stack Empty\n");
        return;
    }
    int x;
    x = S->items[S->top];
    printf("Peeked item is %d\n", x);
}

void print(STACK *S)
{
    if(empty(S)){
        printf("Stack Empty\n");
        return;
    }
    int i;
    for(i = S->top; i >= 0; i--)
        printf("| %d |\n", S->items[i]);
}
```

Queues

Introduction

Real time examples for queues

- Standing in queue to book movie tickets
- Standing in queue to get the tickets/reservation from the ticket counter
- Standing in a queue to get the food in hostel
- Queue the movies/songs in a player

Properties

- Ordered collection of items in which items can be added at one end and deleted from another end
- General convention:
    - Deletion end – front
    - Insertion end – rear
- Order of the Queue is: **First in First out - FIFO (or LILO)**

- Primitive Operations:
    - Insert, remove, and empty
- Insert – operation can always be performed, since there is no limit on number of elements a queue may contain
- Remove – can be applied only when the queue is not empty
- The conventional names used are Enqueue, Dequeue and Display
- The result of an illegal attempt to remove an element from an empty queue is called **underflow**

Implementation Details

We will need three variables:
front and rear → two integer variables
items → array to hold the queue elements

Defining the structure considering above details we have,

```
#define MAXQUEUE  100
struct queue
{
        int front;
        int rear;
        int items[MAXQUEUE];
} q ;
```

Operations:
Initializations: [Done in main( )]

```
q.rear = -1;            // The Insertion end – increment and insert
q.front = 0;            // The deletion end – delete and increment
```

Empty: whenever q.rear < q.front
Note: How did we arrive at this condition?

Si. No.	Queue is empty when	rear value	front vale
1	Initial condition	-1	0
2	Insert 2 elements and delete 2 elements	1	2
3	Insert 4 elements and delete 4 elements	3	4
4	Insert n elements and delete n elements	n-1	n

Hence we can conclude that queue will be empty whenever rear < front index

Full:
Queue will be full when there is no more space left for insertion. As the insertion happens at rear end, we can set the full condition as whenever the rear reaches its maximum index value.
Hence queue will be full when

rear = MAXQUEUE-1

Insert Operation:
insert(q, x)→ Increment rear and insert

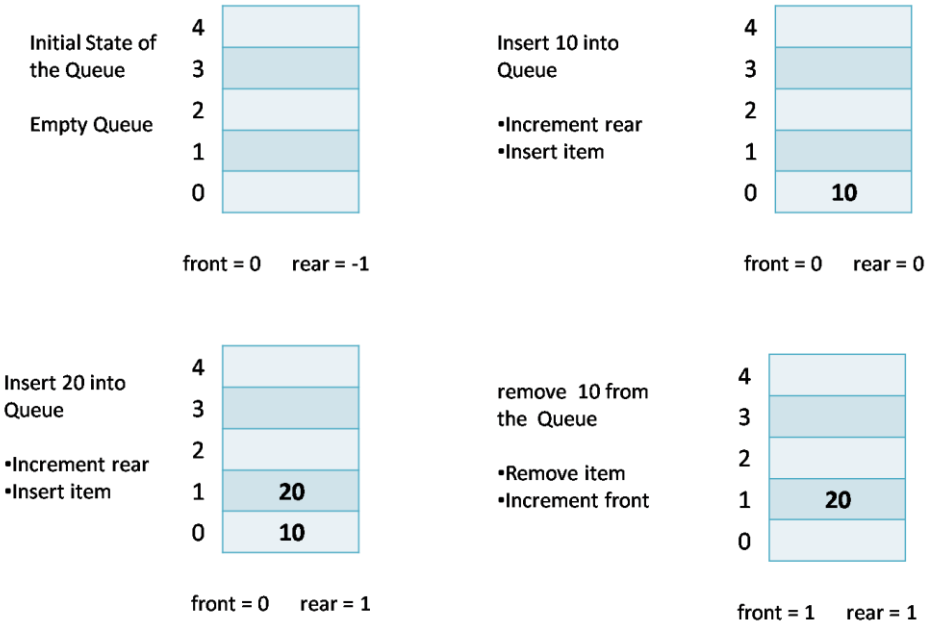
- (q.rear)++;
- q.items[q.rear] = x;

Remove Operation:
x = remove(q) → remove and increment front

- x= q.items[q.front];
- (q.front) ++;

How do you find out the number of elements in a queue?
The number of elements in the queue at any given time is equal to the value of
Number_of_queue_elemets = q.rear – q.front + 1

Queue Sample Operations



Implementation of linear Queue

```c
#include <stdio.h>
#include <stdlib.h>
#define QUEUESIZE 5
#define TRUE 1
#define FALSE 0
```

```c
struct queue
{
    int front;
    int rear;
    int items[QUEUESIZE];
};
typedef struct queue QUEUE;


void enqueue(QUEUE *);
void dequeue(QUEUE *);
void display(QUEUE *);
int full(QUEUE *);
int empty(QUEUE *);

int main()
{
    QUEUE q;
    q.front = 0;
    q.rear = -1;
    int choice;

    while(1){
        printf("MENU\n");
        printf("1-Enqueue\n");
        printf("2-Dequeue\n");
        printf("3-Display\n");
        printf("4-Exit\n");

        printf("\nEnter your choice\n ");
        scanf("%d", &choice);
        switch(choice){
            case 1: enqueue(&q);
                    break;
            case 2: dequeue(&q);
                    break;
            case 3: display(&q);
                    break;
            case 4: printf("Terminating\n");
                    exit(0);
        }
    }
    return 0;
}
```

```
/// Function Name: full
/// Description:   checks if rear end has reached max position
/// Input Param:   Pointer to queue
/// Return type:   TRUE if queue full, FALSE otherwise
int full(QUEUE *q)
{
    if(q->rear == QUEUESIZE - 1)
        return TRUE;
    else
        return FALSE;
}


/// Function Name: enqueuer
/// Description:   enqueue an item inside queue
/// Input Param:   Pointer to queue
/// Return type:   void
void enqueue(QUEUE *q)
{
    if(full(q)){
        printf("Queue full\n");
        return;
    }
    int x;
    printf("Enter the enqueue item\n");
    scanf("%d", &x);
    q->rear++;
    q->items[q->rear] = x;
}


/// Function Name: empty
/// Description:
///                item                    f        r
///                ------------------------------------
///                initial                 0        -1
///                one insertion/deletion  1        0
///                ...
///                n insertions/deletions  n        n-1`
/// Input Param:   Pointer to queue
/// Return type:   TRUE if queue empty, FALSE otherwise
int empty(QUEUE *q)
{
    if(q->front > q->rear)
        return TRUE;
    else
```

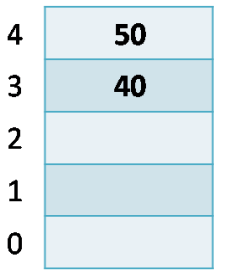
```
        return FALSE;
}


/// Function Name: dequeue
/// Description:   dequeue an item from queue
/// Input Param:   Pointer to queue
/// Return type:   void
void dequeue(QUEUE *q)
{
    if(empty(q)){
        printf("Empty queue\n");
        return;
    }
    int x;
    x = q->items[q->front];
    printf("Dequeued Item is %d\n", x);
    q->front++;
}


/// Function Name: display
/// Description:   display the items from queue
/// Input Param:   Pointer to queue
/// Return type:   void
void display(QUEUE *q)
{
    if(empty(q)){
        printf("Empty Queue\n");
        return;
    }
    int i;
    for(i = q->front; i<= q->rear; i++)
        printf("%d\n", q->items[i]);
}
```

Circular Queue

Even though there is room for new elements in linear queue, at certain conditions we print the message queue is full.
Consider for example, When the Queue size is 5.

Insert 5 items into the Queue and delete 3 items.

Even though there is space for 3 elements, as the rear has reached MAXQUEUE -1 we say Queue is full



front = 3 rear = 4

How to improve?

We can shift all the values one down and make a room for new item which is coming in. However this becomes a costly operation when the size of the queue is very large.
Another possible alternate is to loop back to start of the Queue and continue with insertion again. This defines a **Circular Queue**.

In order to achieve this we make the following changes to the increment operations of rear and front.

```
cq->rear = (cq->rear+1) % MAXQUEUE;
cq->front = (cq->front+1) % MAXQUEUE;
```

Example:

If rear is MAXQUEUE-1, say 4 then,
rear = (4 + 1) % 5 = 5 % 5 = 0.
So the next insertion happens at position 0.
The logic now changes to

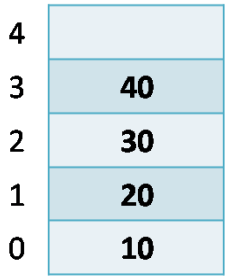
- increment rear and insert and
- increment front and delete

And we always want first insertion to happen at position 0. So we initialize rear and front of the queue with values:
rear = MAXQUEUE – 1
front = MAXQUEUE - 1

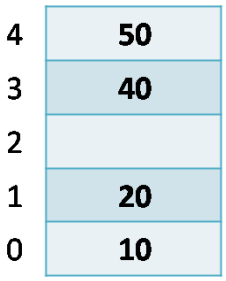
Now with the set initial condition, both empty and full conditions are going to be the same which is front == rear which is not valid.

We keep it as empty condition and design a new condition for full.
According to new design at any given point of time when we say Queue is full, one of the memory locations in items array is left unused.

front == rear + 1 % MAXQUEUE
4 = (3 + 1) % 5
= 4 % 5
= 4



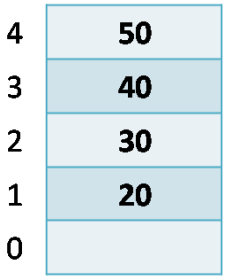
front = 4 rear = 3



front = 2 rear = 1

front == rear + 1 % MAXQUEUE
2 = (1 + 1) % 5
= 2 % 5
= 2

front == rear + 1 % MAXQUEUE
0 = (4 + 1) % 5
= 5 % 5
= 0



front = 0 rear = 4

Above are examples of Queue full. The location pointed by front is always left unused. So when ever front == (rear + 1) % MAXQUEUE we say that Queue is full.

Note:
Can you complete the implementation of circular queue? The code is available for reference under lab exercise section.

Application of Queues

Because of its first in first out nature Queue finds application in many areas.

The basic problem can be viewed as,
In every problem there is a producer and consumer. The production produced by producer is consumed by consumer. If Producer is producing at a very faster rate and consumer has slower consumption rate then we would need an intermediate storage medium to store all the produced items. As the first produced item needs to be consumed first, we would need a queue.

Producer < - - > QUEUE < - - > Consumer

1) Imagine you have a web-site which serves files to thousands of users. You cannot service all requests; you can only handle say 100 at once. A fair policy would be first-come-first

serve: serve 100 at a time in order of arrival. A Queue would definitely be the most appropriate data structure.

2) When a resource is shared among multiple consumers we need a queue
Examples include:

- CPU scheduling that is the way your processor schedules different tasks
- Disk Scheduling (you will study this in your Operating System course).

3) Say you have a number of documents to be printed at once. Your OS puts all of these documents in a queue and sends them to the printer. The printer takes and prints each document in the order the documents are put in the queue, ie, First In, First Out.
In the situation where there are multiple users or a networked computer system, you probably share a printer with other users. When you request to print a file, your request is added to the print queue. When your request reaches the front of the print queue, your file is printed. This ensures that only one person at a time has access to the printer and that this access is given on a first-come, first-served basis.

4) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes we would need a queue to store.
Examples include IO Buffers, pipes, file IO, etc. (you will study this in your Operating System and Networking courses)

~*~*~*~*~*~*~