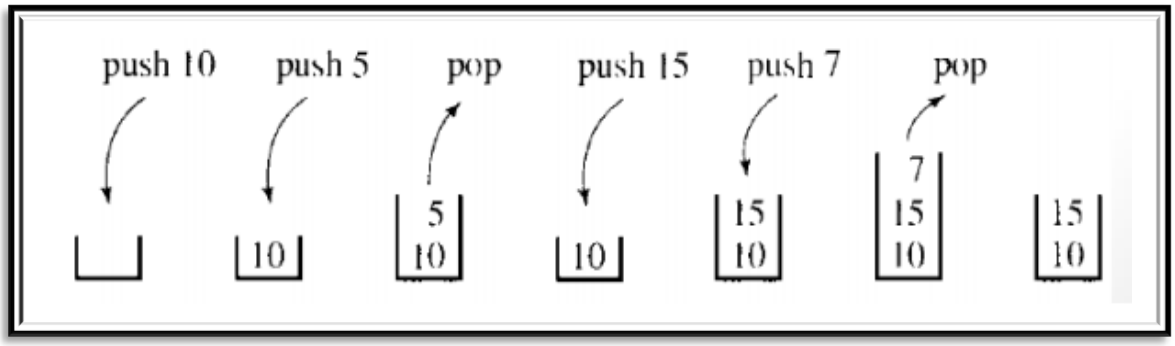
Stack

Overview

- **A stack is Linear - non-primitive data structure.**
- **The stack is called Last-in-First-out (LIFO).**
- **You push in a stack and pop the last pushed element.**

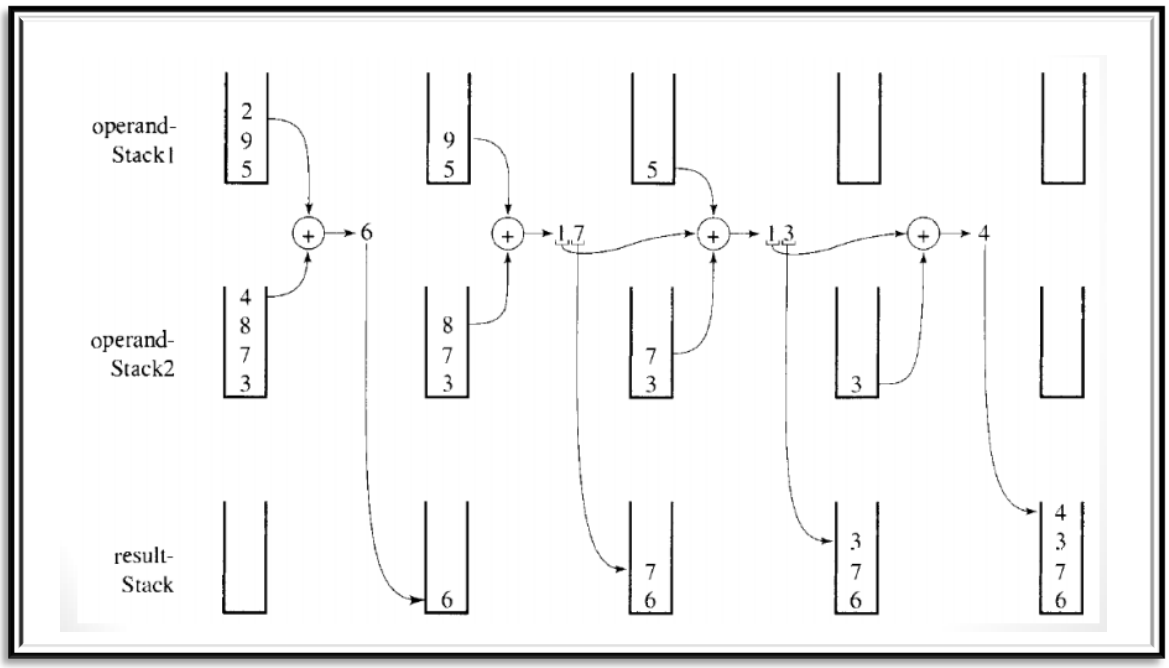


Stack Apps

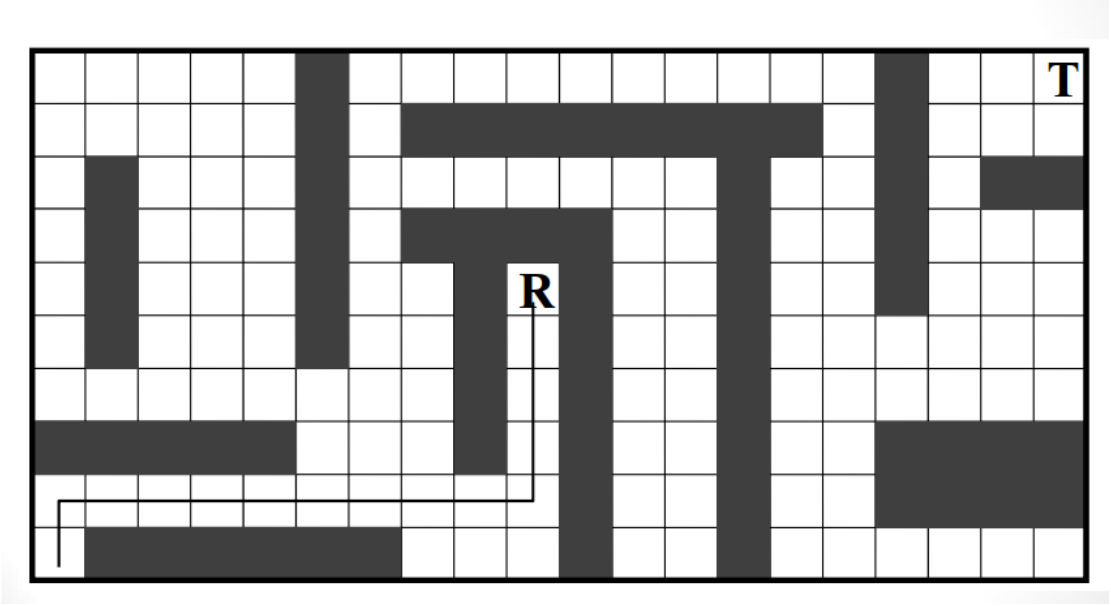
- **Stacks can be used to check parenthesis matching in an expression.**
- **Reverse a word/string**
- **"undo" mechanism in text editors**
- **mathematical expressions evaluation**
- **Stacks can be used for system stack (stack memory)**
- **Stack data structures are used in backtracking problems (Robot Navigation Problem)**

Apps Examples

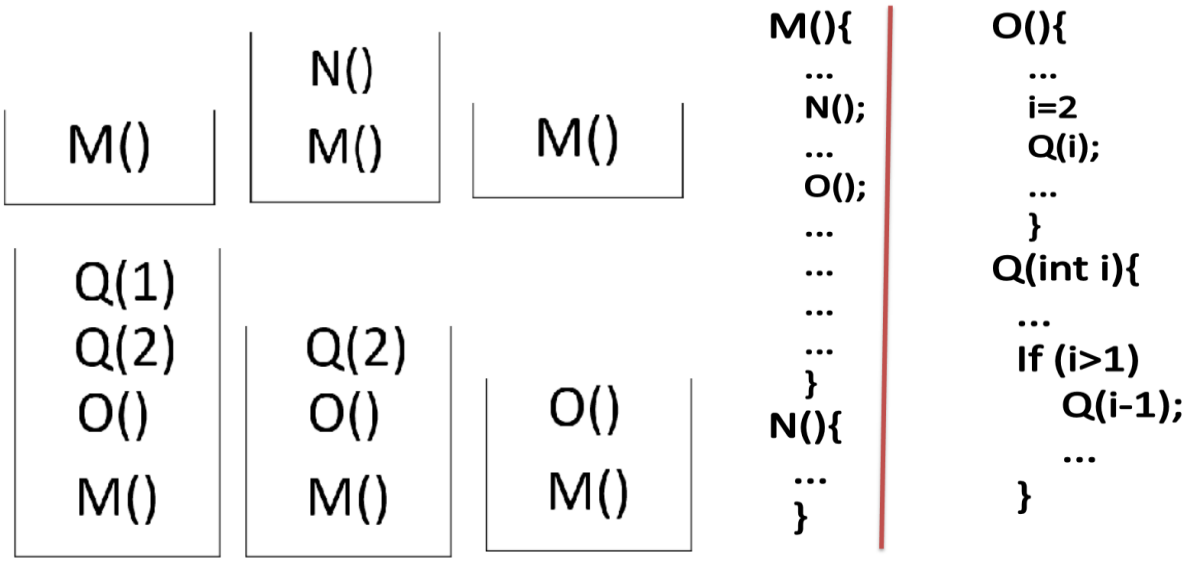
- **Adding Large Numbers. *E.g.* 592+3784**



- **Backtracking robot movements when it reaches a dead path.**



System Stack



Stack Operations

- **Initialization:** Create the stack, leaving it empty.
  - **Pre:** None.
  - **Post:** The stack is initialized to be empty.
- **Empty operation:** Determine whether the stack is empty or not.
  - **Pre:** The stack is initialized.
  - **Post:** If the stack is empty (1) is returned. Otherwise (0) is returned
- **Full operation:** Determine whether the stack is full or not.
  - **Pre:** The stack is initialized.
  - **Post:** If the stack is full (1) is returned. Otherwise (0) is returned.
- **Push operation:** Push a new entry onto the top of the stack
  - **Pre:** The stack is initialized.
  - **Post:** If the stack is not full, item is added to the top of the stack. Otherwise, an error message is displayed and the stack is left unchanged

- **Pop operation:** Pop the entry off the top of the stack.
  - **Pre:** The stack is initialized.
  - **Post:** If the stack is not empty the top element of the stack is removed from it and is assigned to item. Otherwise, an error message is displayed and the stack is left unchanged

Stack.h. File

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h> // for strlen function
#define MAX 10
typedef char EntryType;
typedef struct{
    int top;
    EntryType stackArray[ MAX ];
} StackType;
void  CreateStack(StackType *s);
bool StackEmpty(StackType s);
bool StackFull(StackType s);
void Push(EntryType  item, StackType *s);
void Pop(EntryType*item, StackType*s);
void Peek(EntryType*item, StackType*s);
void PrintStack(StackType*s);
```

Stack.C. File

```
#include "static_stack.h"
/*
Pre: None.
Post: The stack is initialized to be empty.
*/
void  CreateStack(StackType *s){
    s->top = -1;
}
```

```
/*
Pre: The stack is initialized.
Post: If the stack is empty (1) is returned. Otherwise (0) is
returned.
*/
bool StackEmpty(StackType s){
    return (s.top == -1);
}
/*
Pre: The stack is initialized.
Post: If the stack is full (1) is returned. Otherwise (0) is
returned.
*/
bool StackFull(StackType s){
    return (s.top==MAX-1);
}
/*
Pre: The stack is initialized.
Post: If the stack is not full, item is added to the top of the
stack. Otherwise, an error message is displayed and the
stack is left unchanged
*/
void Push(EntryType  item, StackType *s){

    if (StackFull(*s))
        printf("Error: Stack Overflow \n");
    else
    {
        s->top++;
        s->stackArray[s->top]=item;
    }

}
/*
Pre: The stack is initialized.
Post: If the stack is not empty The top element of the stack is
removed from it and is assigned to item. Otherwise, an error
message is displayed and the stack is left unchanged
*/
void Pop(EntryType*item, StackType*s){
    if (StackEmpty(*s))
        printf("Error: Stack is empty \n");
    else
        *item = s->stackArray[s->top--];
}
```

```
/*
Pre: The stack is initialized.
Post: If the stack is not empty The top element of the stack is
assigned to item without removing. Otherwise, an error
message is displayed and the stack is left unchanged
*/
void Peek(EntryType*item, StackType*s){

    if (StackEmpty(*s))
        printf("Error: Stack is empty \n");
    else
        *item = s->stackArray[s->top];
}

void PrintStack(StackType*s){
    if (StackEmpty(*s))
        printf("Error: Stack is empty \n");
    else
    {
        int i;
        for(i=s->top ; i>-1 ; i--)
            printf("%d \n",s->stackArray[i]);
    }
}
```

Exercise

Q1: Assume that we need to read a line of text and write it back in a reverse order.

Answer

```
StackType stack;
//Initialize the stack to be empty
CreateStack(&stack);

item = getchar();
```

While

```
!StackFull(stack)&& item!= '\n'){
    //Push each item onto the stack
    Push(item, &stack);
    item = getchar();
}
```

While

```
(!StackEmpty(stack)){
    //Pop an item from the stack
    Pop(&item, &stack);
    putchar(item);
  }
```

Q2: As a user for the stack ADT, write the StackTop function which return the top element of the stack and left the stack unchanged (user level and implementation level)

Answer

```
EntryType   StackTop(StackType  * s){
       EntryType   item;
       pop(&item,  s);
      push(item,  s);
      return (item);
}
```

Sheet 2

1. Write a function that returns the first element entered to a stack. (Implementation level)

Answer

```
void GetFirstElement(EntryType*item, StackType s){
    *item = s.stackArray[0];
}

EntryType GetFirst(StackType s) {
    return s.stackArray[0];
}
```

2. Write a function that returns a copy from the last element in a stack. (Implementation level)

Answer

```
void Peek(EntryType*item, StackType*s){

    if (StackEmpty(*s))
        printf("Error: Stack is empty \n");
    else
        *item = s->stackArray[s->top];
}
```

3. Write a function to destroy a stack. (Implementation level)

Answer

```
void DestroyStack(StackType *s){
    s->top = -1;
}
```

4. Write a function to copy a stack to another.
(Implementation level)

Answer

```
void CopyStack(StackType s_orig,StackType *s_copy){
    unsigned int i ;
    for(i = 0; i <= s_orig.top; i++)
        s_copy->stackArray[++s_copy->top] = s_orig.stackArray[i];
}
```

5. Write a function to return the size of a stack
(implementation level)

Answer

```
unsigned int GetSize(StackType s){
    return (s.top) +1 ;
}
```

6. Write a function that returns the first element entered to a stack. (User level)

Answer

```
EntryType GetFirstElement_userlevel(StackType *s){
     StackType temp;
     CreateStack(&temp);
     EntryType item;
     while(!StackEmpty(*s)){
          Pop(&item,s);
          Push(item,&temp);
     }

     EntryType retv = item;
     while(!StackEmpty(temp)){
          Pop(&item,&temp);
          Push(item,s);
     }

     return retv;
}
```

7. Write a function that returns a copy from the last element in a stack. (User level)

Answer

```
 EntryType GetLastElem_userlevel(StackType *s)
}{
     EntryType retv;
     Pop(&retv,s);
     Push(retv,s);
     return retv;
-}
```

8. Write a function to destroy a stack. (User level)

Answer

```
void DestroyStack_userlevel(StackType *s){
    EntryType item;
    while(!StackEmpty(*s)){
        Pop(&item,s);
    }
}
```

9. Write a function to copy a stack to another. (User level)

Answer

```
void CopyStack_userlevel(StackType s_orig,StackType *s_copy){

    StackType temp;
    CreateStack(&temp);
    EntryType item;
    while(!StackEmpty(s_orig)){
        Pop(&item,&s_orig);
        Push(item,&temp);
    }
    while(!StackEmpty(temp)){
        Pop(&item,&temp);
        Push(item,s_copy);
    }
}
```

10. Write a function to return the size of a stack (user level)

Answer

```
unsigned int GetSize_userlevel(StackType s){
    EntryType item;
    unsigned int count = 0;
    while(!StackEmpty(s)){
        Pop(&item,&s);
        count++;
    }
    return count;
}
```

Sheet 3

1. Write a function that returns the last element in a queue. (implementation level)

Answer

```
EntryType GetLast(QueueType q){

    if (QueueEmpty(q))
        printf("Error: Queue is empty \n");
    else
        return q.queueArray[q.rear];
}
```

2. Write a function that returns a copy from the first element in a queue. (implementation level)

Answer

```
EntryType GetFirst(QueueType q){

    if (QueueEmpty(q))
        printf("Error: Queue is empty \n");
    else
        return q.queueArray[q.front];
}
```

3. Write a function to destroy a queue (implementation level)

Answer

```
void DestroyQueue(QueueType *q){
    q->front= 0;
    q->rear = MAX -1;
    q->size = 0;
}
```

4. Write a function to copy a queue to another. (implementation level)

Answer

```
void CopyQueue(QueueType q_orig,QueueType *q_copy){
    q_copy->front = q_orig.front;
    q_copy->rear  = q_orig.rear;
    q_copy->size  = q_orig.size;
    int i, counter;
    for(i=q_orig.front, counter=0; counter < q_orig.size; counter++){
        q_copy->queueArray[i] = q_orig.queueArray[i];
        i=(i+1) % MAX;
    }
}
```

5. Write a function to return the size of a queue (implementation level)

Answer

```
unsigned int GetSize(QueueType q){
    return  q.size;
}
```

6. Write a function that returns the last element in a queue. (user level)

Answer

```
EntryType GetLast_userlevel(QueueType q){
    EntryType retv;
    while(!QueueEmpty(q))
        Dequeue(&retv,&q);

    return retv;
}
```

7. Write a function that returns a copy from the first element in a queue. (user level)

Answer

```
EntryType GetFirst_userlevel(QueueType q){
    EntryType retv;
    Dequeue(&retv,&q);
    return retv;
}
```

8. Write a function to destroy a queue (user level)

Answer

```
void DestroyQueue_userlevel(QueueType *q){
     EntryType item;
     while(!QueueEmpty(*q)){
          Dequeue(&item,q);
     }
}
```

9. Write a function to copy a queue to another. (user level)

Answer

```
void CopyQueue_userlevel(QueueType *s_orig,QueueType *s_copy){
     EntryType item;
     QueueType temp;
     CreateQueue(&temp);
     while(!QueueEmpty(*s_orig)){
          Dequeue(&item,s_orig);
          Enqueue(item,s_copy);
          Enqueue(item,&temp);
     }

     while(!QueueEmpty(temp)){
          Dequeue(&item,&temp);
          Enqueue(item,s_orig);
     }
}
```

10. Write a function to return the size of a queue (user level)

Answer

Idea queue passed by value, loop until empty and inside the loop dequeue all elements and increment a counter then after loop finishes, return the counter

11. We (as a user for QueueADT) have two filled queues; the first queue holds section code while the other holds group code (where number of groups inside the section is maximum10). Merge those numbers (section code*10+group code) in a newly created queue)

Answer

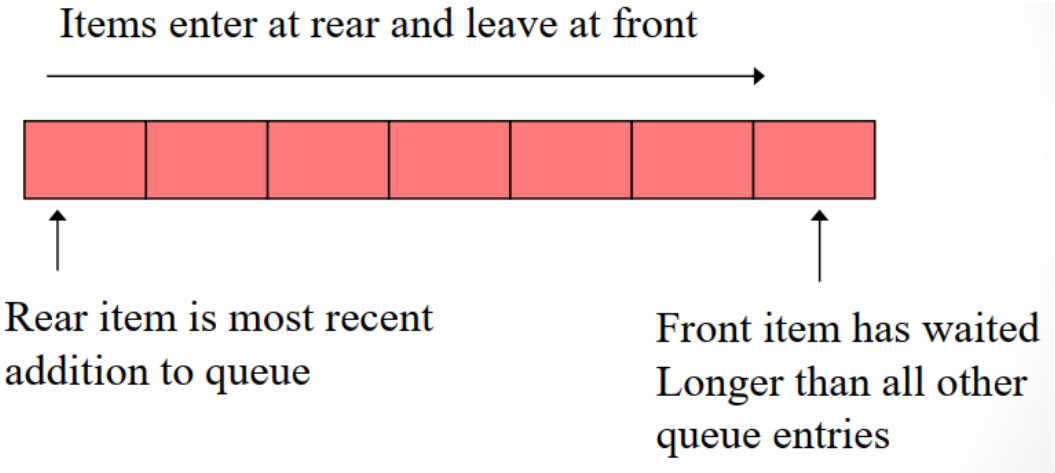
```
QueueType s;
QueueType g;
QueueType mix;
CreateQueue(&s);
CreateQueue(&g);
CreateQueue(&mix);
Enqueue(1,&s);
Enqueue(2,&s);
Enqueue(3,&s);
Enqueue(4,&s);
Enqueue(5,&s);
Enqueue(10,&g);
Enqueue(20,&g);
Enqueue(30,&g);
Enqueue(40,&g);
Enqueue(50,&g);

Enqueue(Dequeue(&s)*10 + Dequeue(&g),&mix);
Enqueue(Dequeue(&s)*10 + Dequeue(&g),&mix);
Enqueue(Dequeue(&s)*10 + Dequeue(&g),&mix);
Enqueue(Dequeue(&s)*10 + Dequeue(&g),&mix);
Enqueue(Dequeue(&s)*10 + Dequeue(&g),&mix);
```

Queue

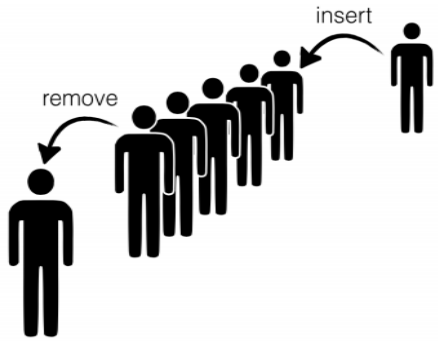
Overview

- A queue is Linear - non-primitive data structure.
- The queue is called First-in-First-out (FIFO)
- New elements are added at one end called rear, and the existing elements are deleted from other end called front.
- Queues have the property that, the earlier an item enters a queue, the earlier it will leave the queue.



Queue Apps

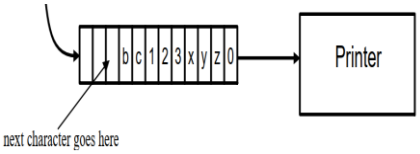
- **Real world examples**
  - Cars in line at a tool booth
  - People waiting in line for a movie



- **Computer world examples**
  - Characters waiting to be printed
  - Jobs waiting to be executed

Queues and lists

- A queue is a restricted form of a list.
- Additions to the queue must occur at the rear.
- Deletions from the queue must occur at the front



Queue operations

- **Initialization:** Create the Queue, leaving it empty.
- **Pre:** None.
- **Post:** The queue is initialized to be empty.
- **Empty operation:** Determine whether the Queue is empty or not.
- **Pre:** The queue is initialized.
- **Post:** If the queue is empty (1) is returned. Otherwise (0) is returned
- **Full operation:** Determine whether the Queue is full or not.
- Pre: The queue is initialized.
- Post: If the queue is full (1) is returned. Otherwise (0) is returned.
- **Enqueue operation:** Push a new entry onto the end of the queue
- **Pre:** The queue is initialized and is not full.
- **Post:** Item is added to the end of the queue.

- **Dequeue operation:** remove element at the head of the queue

➢ **Pre:** The Queue is initialized and is not empty.

➢ **Post:** The front element of the Queue is removed from it and is assigned to item.

Queue implementation

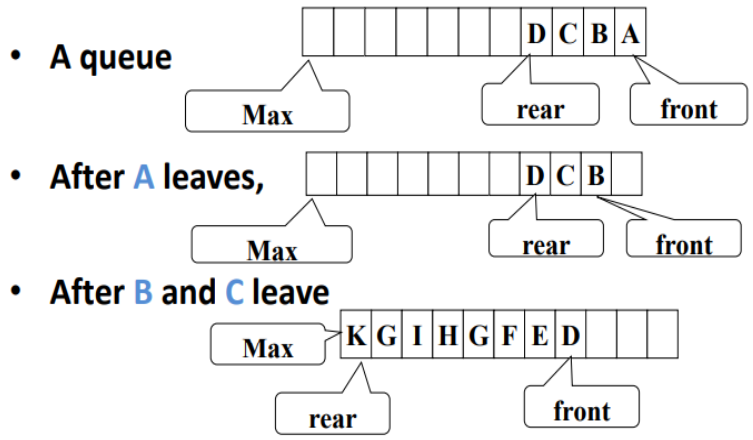
With data in a queue, implemented as an array, rear will normally be greater than front.

THERE ARE 3 SPECIAL SITUATIONS

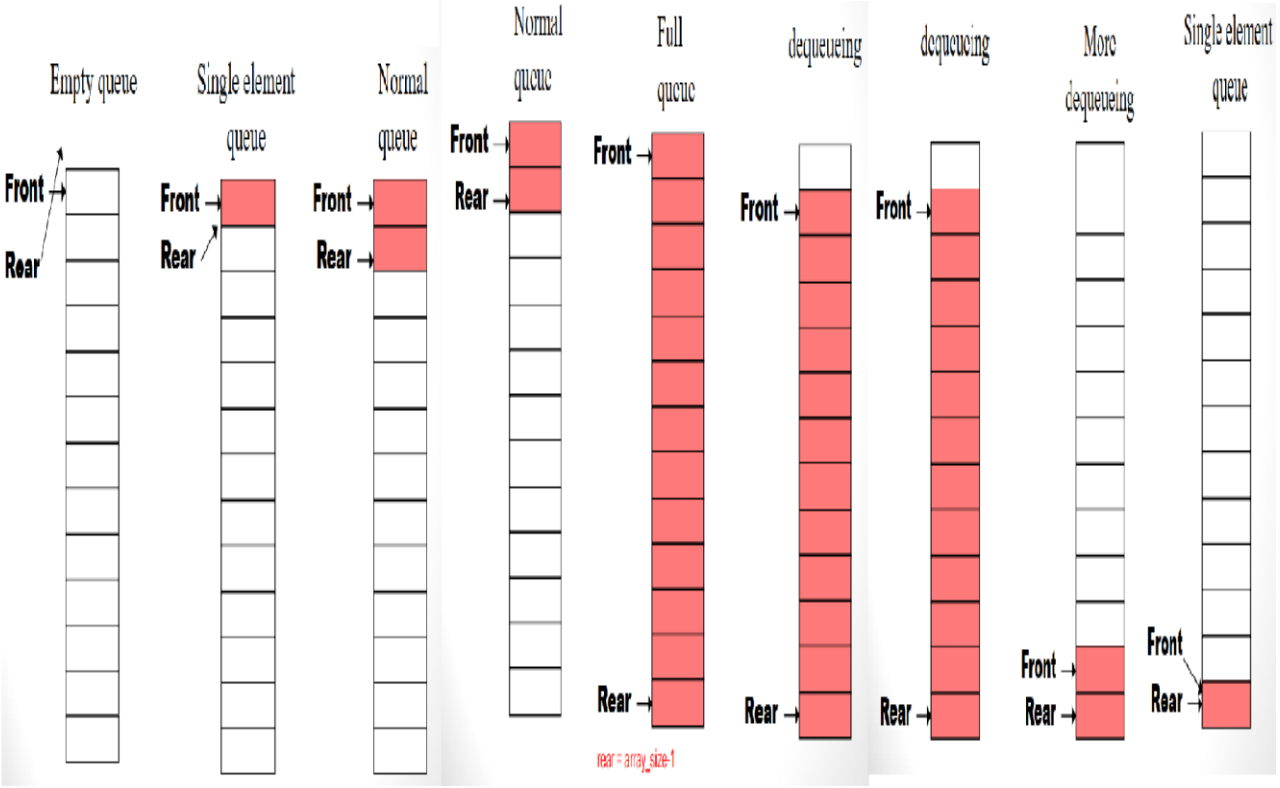
1. Rear < front (empty queue)
2. Rear = front (one-entry queue)
3. Rear = array size (full queue)

➢ We cannot increase rear beyond the limits of the array.

➢ Therefore, this queue must be regarded as full, even though there are plenty of available memory cells.

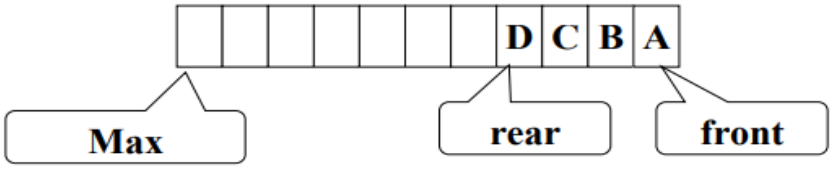


How can we insert a new element?!!

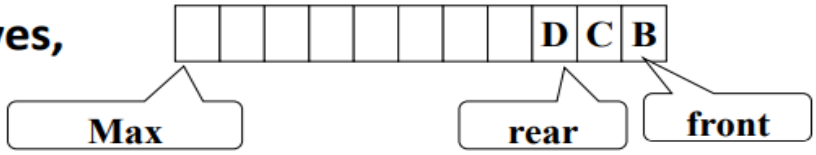


Problem Solution 1

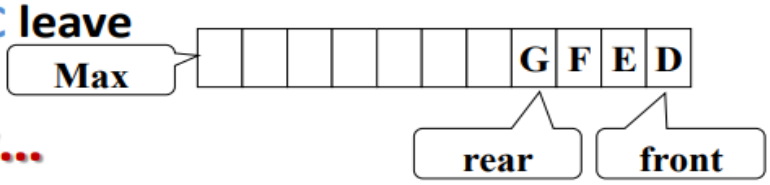
One solution is to shifting all items to front when dequeue operation.



After A leaves,



After B and C leave



Too Costly...

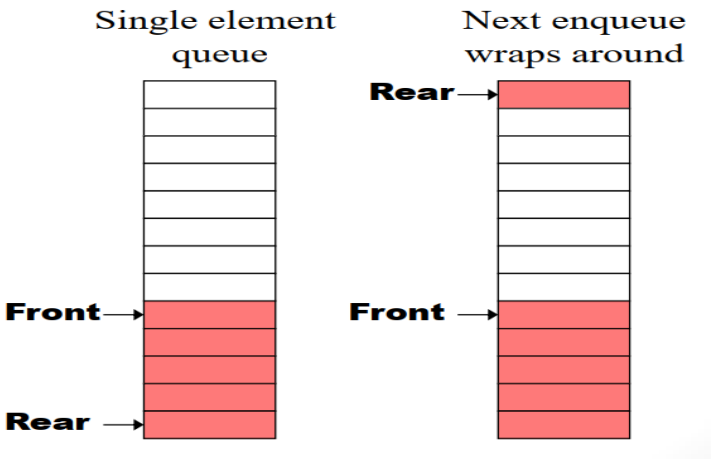
Problem Solution 2

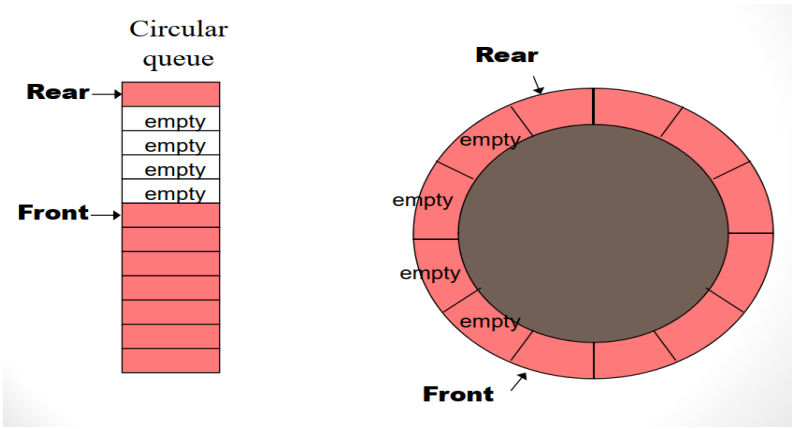
To get around this, we have to make it possible to '**wrap around**'

Both 'rear' and 'front' must be able to wrap around

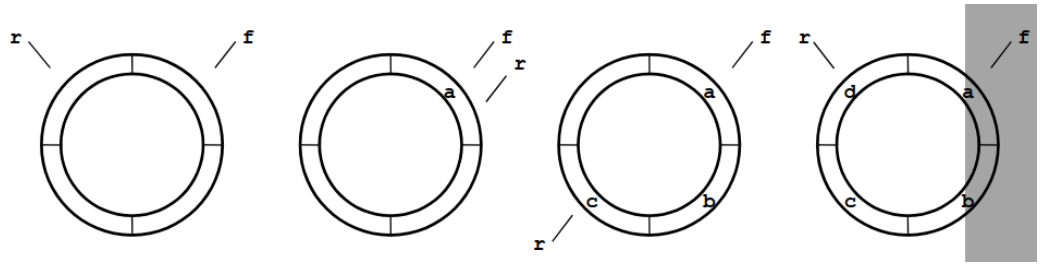
If rear + 1 > maxQueue -1 then set rear to 0

rear = (rear+1) % maxQueue





Circular Queue and Wrapping around issue FULL or EMPTY???



- **In both cases front comes after rear with one step. Then, how can we distinguish between the two cases?!!**
- **One solution is to keep track with the number of elements on each queue.**

Queue. h File

```
#include <stdio.h>
#include <stdbool.h>

#define MAX 10
typedef unsigned int EntryType;
typedef   struct{
    int front;
    int rear;
    int size;
    EntryType queueArray[MAX];
} QueueType;

void CreateQueue(QueueType *q);
bool QueueEmpty(QueueType q);
bool QueueFull(QueueType q);
void Enqueue(EntryType  item, QueueType *q);
void Dequeue(EntryType* item, QueueType *q);
void Front(EntryType* item, QueueType q);
void TraverseQueue(QueueType q);
```

Queue. c File

```
void CreateQueue(QueueType *q){
    q->front= 0;
    q->rear = MAX -1;
    q->size = 0;
}

bool QueueEmpty(QueueType q){
    return (q.size == 0);
}

bool QueueFull(QueueType q){
    return (q.size == MAX);
}

void Enqueue(EntryType  item, QueueType *q){

    if (QueueFull(*q))
        printf("Error: Queue Overflow \n");
    else
    {
        q->rear = (q->rear+1) % MAX;
        // to circulate the queue
        q->queueArray[q->rear] = item;
        q->size ++;
    }

}

void Dequeue(EntryType* item, QueueType *q){
    if (QueueEmpty(*q))
        printf("Error: Queue is empty \n");
    else
    {
        *item = q->queueArray[q->front];
        q->front = (q->front +1) % MAX;
        q->size--;
    }
}
```

```
void Front(EntryType* item, QueueType q){
    if (QueueEmpty(q))
        printf("Error: Queue is empty \n");
    else
    {
        *item = q.queueArray[q.front];
    }
}

void TraverseQueue(QueueType q){
    int i, counter;
    for(i=q.front, counter=0; counter < q.size; counter++){
       printf("Queue[%d] = %d\n", i ,q.queueArray[i]);
        i=(i+1) % MAX;
    }
}
```