MODULE 2: STACKS AND QUEUES

STACKS

DEFINITION

"A **stack** is an **ordered list** in which insertions (pushes) and deletions (pops) are made at one end called the **top**."

Given a stack **S= (a_0, ... ,a_{n-1})**, where **a_0** is the bottom element, **a_{n-1}** is the top element, and **a_i** is on top of element **a_{i-1}**, **0 < i < n**.

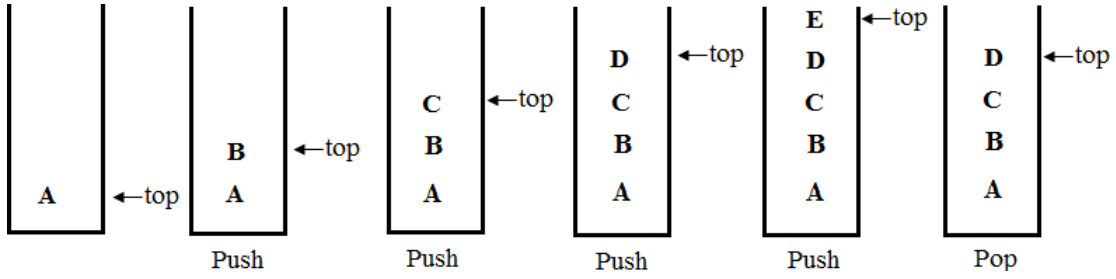


Figure: Inserting and deleting elements in a stack

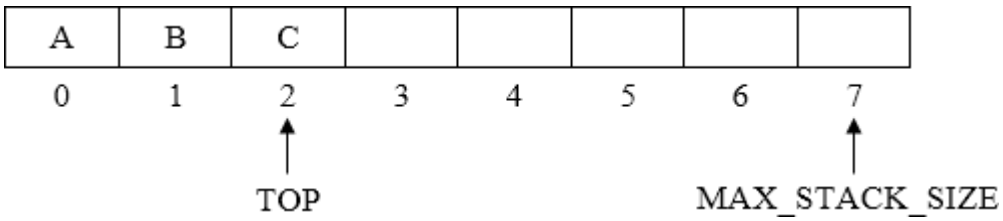
As shown in above figure, the elements are added in the stack in the order A, B, C, D, E, then **E** is the first element that is deleted from the stack and the last element is deleted from stack is **A**. Figure illustrates this sequence of operations.

Since the last element inserted into a stack is the first element removed, a stack is also known as a **Last-In-First-Out (LIFO)** list.

ARRAY REPRESENTATION OF STACKS

- Stacks may be represented in the computer in various ways such as one-way linked list (Singly linked list) or linear array.
- Stacks are maintained by the two variables such as TOP and MAX_STACK_SIZE.
- TOP which contains the location of the top element in the stack. If TOP= -1, then it indicates stack is empty.
- MAX_STACK_SIZE which gives maximum number of elements that can be stored in stack.

Stack can represented using linear array as shown below



STACK OPERATIONS

Implementation of the stack operations as follows.

1. Stack Create

```
Stack CreateS(maxStackSize )::=
    #define MAX_STACK_SIZE 100 /* maximum stack size*/
    typedef struct
        {
            int key;
                        /* other fields */
        } element;
    element stack[MAX_STACK_SIZE];
    int top = -1;
```

The **element** which is used to insert or delete is specified as a structure that consists of only a **key** field.

2. Boolean IsEmpty(Stack)::= top < 0;

3. Boolean IsFull(Stack)::= top >= MAX_STACK_SIZE-1;

The I**sEmpty** and **IsFull** operations are simple, and is implemented directly in the program push and pop functions. Each of these functions assumes that the variables **stack** and **top** are global.

4. Push( )

Function **push** checks whether stack is full. If it is, it calls stackFull( ), which prints an error message and terminates execution. When the stack is not full, increment top and assign item to stack [top].

```
    void push(element item)
        { /* add an item to the global stack */
            if (top >= MAX_STACK_SIZE-1)
                stackFull();
            stack[++top] = item;
        }
```

5. Pop( )

Deleting an element from the stack is called pop operation. The element is deleted only from the top of the stack and only one element is deleted at a time.

```
element pop ( )
        { /*delete and return the top element from the stack */
                if (top == -1)
                        return stackEmpty(); /*returns an error key */
                return stack[top--];
        }
```

6. stackFull( )

The **stackFull** which prints an error message and terminates execution.

```
void stackFull()
        {
                fprintf(stderr, "Stack is full, cannot add element");
                exit(EXIT_FAILURE);
        }
```

STACKS USING DYNAMIC ARRAYS

The array is used to implement stack, but the bound (MAX_STACK_SIZE) should be known during compile time. The size of bound is impossible to alter during compilation hence this can be overcome by using dynamically allocated array for the elements and then increasing the size of array as needed.

Stack Operations using dynamic array

1. Stack CreateS( )::=

```
typedef struct
        {
                int key;        /* other fields */
        } element;
element *stack;
MALLOC(stack, sizeof(*stack));
int capacity= 1;
int top= -1;
```

2. Boolean IsEmpty(Stack)::= top < 0;

3. Boolean IsFull(Stack)::= top >= capacity-1;

4. push()

Here the MAX_STACK_SIZE is replaced with **capacity**

```
void push(element item)
    {   /* add an item to the global stack */
        if (top >= capacity-1)
            stackFull();
        stack[++top] = item;
    }
```

5. pop( )

In this function, no changes are made.

```
element pop ( )
    {   /* delete and return the top element from the stack */
        if (top == -1)
            return stackEmpty(); /* returns an error key */
        return stack[top--];
    }
```

6. stackFull( )

The new code shown below, attempts to increase the **capacity** of the array **stack** so that new element can be added into the stack. Before increasing the capacity of an array, decide what the new capacity should be.

In array doubling, array capacity is doubled whenever it becomes necessary to increase the capacity of an array.

```
void stackFull()
{
    REALLOC (stack, 2*capacity*sizeof(*stack));
    capacity *= 2;
}
```

Stack full with array doubling

Analysis

In the **worst case**, the realloc function needs to allocate **2*capacity*sizeof (*stack)** bytes of memory and copy **capacity *sizeof (*stack))** bytes of memory from the old array into the new one. Under the assumptions that memory may be allocated in O(1) time and that a stack element can be copied in O(1) time, the time required by array doubling is O(capacity). Initially, capacity is 1.

Suppose that, if all elements are pushed in stack and the capacity is 2k for some k, k>O, then the total time spent over all array doublings is $\mathbf{O\left(\sum_{i=1}^{k} 2i\right) = O(2^{k+l}) = O(2^{k})}$.

Since the total number of pushes is more than **2k-1**, the total time spend in array doubling is **O(n)**, where **n** is the total number of pushes. Hence, even with the time spent on array doubling added in, the total run time of push over all **n** pushes is **O(n).**

STACK APPLICATIONS: POLISH NOTATION

Expressions: It is sequence of operators and operands that reduces to a single value after evaluation is called an expression.

X = a / b – c + d * e – a * c

In above expression contains operators (+, –, /, *) operands (a, b, c, d, e).

Expression can be represented in in different format such as

- Prefix Expression or Polish notation
- Infix Expression
- Postfix Expression or Reverse Polish notation

Infix Expression: In this expression, the binary operator is placed in-between the operand. The expression can be parenthesized or un- parenthesized.

Example: A + B

Here, **A** & **B** are operands and **+** is operand

Prefix or Polish Expression: In this expression, the operator appears before its operand.

Example: + A B

Here, **A** & **B** are operands and **+** is operand

Postfix or Reverse Polish Expression: In this expression, the operator appears after its operand.

Example: A B +

Here, **A** & **B** are operands and **+** is operand

Precedence of the operators

The first problem with understanding the meaning of expressions and statements is finding out the order in which the operations are performed.

Example: assume that a =4, b =c =2, d =e =3 in below expression

X = a / b – c + d * e – a * c

((4/2)-2) + (3*3)-(4*2)
=0+9-8
=1

OR

(4/ (2-2 +3)) *(3-4)*2
= (4/3) * (-1) * 2
= -2.66666

The first answer is picked most because division is carried out before subtraction, and multiplication before addition. If we wanted the second answer, write expression differently using parentheses to change the order of evaluation

X= ((a / (b – c + d)) * (e – a) * c

In C, there is a precedence hierarchy that determines the order in which operators are evaluated. Below figure contains the precedence hierarchy for C.

Token	Operator	Precedence	Associativity
( ) [ ] ⟶	function call array element struct or union member	17	left-to-right
-- ++	Increment, Decrement	16	left-to-right
--++ ! ~ -+ & * sizeof	decrement, increment logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	Multiplicative	13	left-to-right
+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
== !=	equality	9	left-to-right
&	Bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
|	Bitwise or	6	left-to-right
&&	logical and	5	left-to-right
||	logical or	4	left-to-right
?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^= |=	assignment	2	right -to-left
,	comma	1	left-to-right

- The operators are arranged from highest precedence to lowest. Operators with highest precedence are evaluated first.
- The associativity column indicates how to evaluate operators with the same precedence. For example, the multiplicative operators have left-to-right associativity. This means that the expression **a * b / c % d / e** is equivalent to **((((a * b) / c) % d) / e)**
- Parentheses are used to override precedence, and expressions are always evaluated from the innermost parenthesized expression first

INFIX TO POSTFIX CONVERSION

An algorithm to convert infix to a postfix expression as follows:

1. Fully parenthesize the expression.
2. Move all binary operators so that they replace their corresponding right parentheses.
3. Delete all parentheses.

Example: Infix expression: a/b -c +d*e -a*c
Fully parenthesized : ((((a/b)-c) + (d*e))-a*c))
: a b / e – d e * + a c *

Example [Parenthesized expression]: Parentheses make the translation process more difficult because the equivalent postfix expression will be parenthesis-free.
The expression **a*(b +c)*d** which results **abc +*d*** in postfix. Figure shows the translation process.

Token	Stack [0]	[1]	[2]	Top	Output
a				-1	a
*	*			0	a
(	*	(		1	a
b	*	(		1	ab
+	*	(	+	2	ab
c	*	(	+	2	abc
)	*			0	abc+
*	*			0	abc +*
d	*			0	abc +*d
eos	*			0	abc +*d*

- The analysis of the examples suggests a precedence-based scheme for stacking and unstacking operators.
- The left parenthesis complicates matters because it behaves like a low-precedence operator when it is on the stack and a high-precedence one when it is not. It is placed in the stack whenever it is found in the expression, but it is unstacked only when its matching right parenthesis is found.
- There are two types of precedence, **in-stack precedence (isp)** and **incoming precedence (icp)**.

QUEUES

DEFINITION

- "A queue is an **ordered list** in which insertions (additions, pushes) and deletions (removals and pops) take place at different ends."
- The end at which new elements are added is called the **rear**, and that from which old elements are deleted is called the **front**.

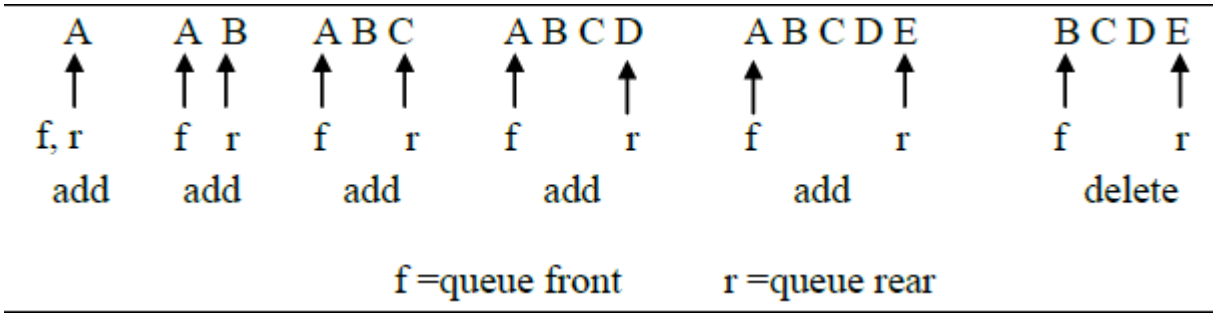
If the elements are inserted A, B, C, D and E in this order, then A is the first element deleted from the queue. Since the first element inserted into a queue is the first element removed, queues are also known as **First-In-First-Out (FIFO) lists.**

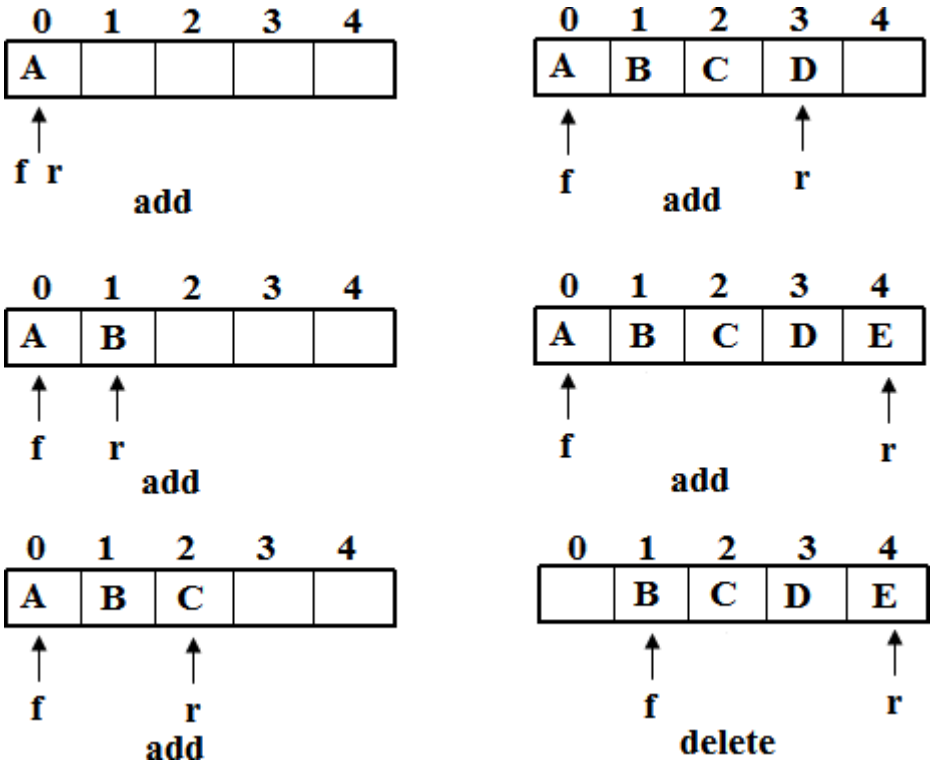
QUEUE REPRESENTATION USING ARRAY

- Queues may be represented by one-way lists or linear arrays.
- Queues will be maintained by a linear array QUEUE and two pointer variables:
  FRONT-containing the location of the front element of the queue
  REAR-containing the location of the rear element of the queue.
- The condition FRONT = NULL will indicate that the queue is empty.

Figure indicates the way elements will be deleted from the queue and the way new elements will be added to the queue.

- Whenever an element is deleted from the queue, the value of FRONT is increased by 1; this can be implemented by the assignment FRONT := FRONT + 1
- When an element is added to the queue, the value of REAR is increased by 1; this can be implemented by the assignment REAR := REAR + 1





QUEUE OPERATIONS

Implementation of the queue operations as follows.

1. Queue Create

```
Queue CreateQ(maxQueueSize) ::=
        #define MAX_QUEUE_SIZE  100          /* maximum queue size */
        typedef struct
                {
                        int key;             /* other fields */
                } element;
        element queue[MAX_QUEUE_SIZE];
        int rear = -1;
        int front = -1;
```

2. Boolean IsEmptyQ(queue) ::= front ==rear

3. Boolean IsFullQ(queue) ::= rear == MAX_QUEUE_SIZE-1

In the queue, two variables are used which are **front** and **rear.** The queue increments **rear** in addq( ) and **front** in delete( ). The function calls would be
addq (item); and **item =delete( );**

4. addq(item)

```
void addq(element item)
        {                       /* add an item to the queue */
                if (rear == MAX_QUEUE_SIZE-1)
                        queueFull();
                queue [++rear] = item;
        }
```

Program: Add to a queue

5. deleteq( )

```
element deleteq()
        {                       /* remove element at the front of the queue */
                if (front == rear)
                        return queueEmpty( );           /* return an error key */
                return queue[++front];
        }
```

Program: Delete from a queue

6. queueFull( )

The **queueFull** function which prints an error message and terminates execution

```
void queueFull()
        {
                fprintf(stderr, "Queue is full, cannot add element");
                exit(EXIT_FAILURE);
        }
```


Example: Job scheduling

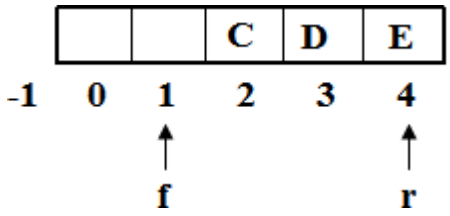
- Queues are frequently used in creation of a **job queue** by an operating system. If the operating system does not use priorities, then the jobs are processed in the order they enter the system.
- Figure illustrates how an operating system process jobs using a sequential representation for its queue.

front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Comments
-1	-1					Queue is empty
-1	0	J1				Job 1 is added
-1	1	J1	J2			Job 2 is added
-1	2	J1	J2	J3		Job 3 is added
0	2		J2	J3		Job 1 is deleted
1	2			J3		Job 2 is deleted

Figure: Insertion and deletion from a sequential queue

Drawback of Queue

When item enters and deleted from the queue, the queue gradually shifts to the right as shown in figure.

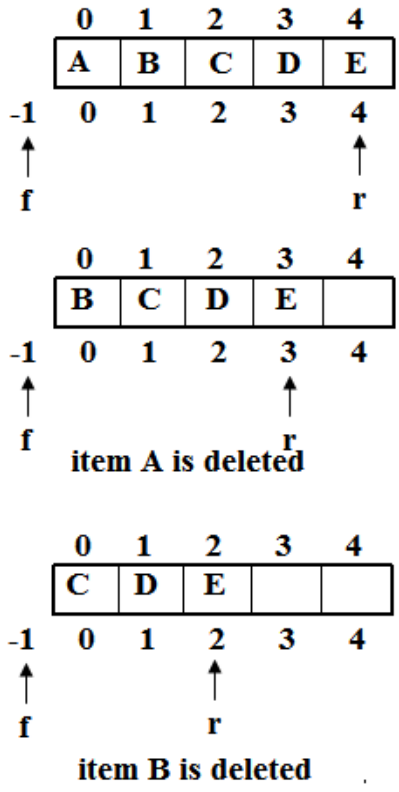


In this above situation, when we try to insert another item, which shows that the **queue is full** . This means that the **rear** index equals to MAX_QUEUE_SIZE -1. But even if the space is available at the front end, rear insertion cannot be done.

Overcome of Drawback using different methods

Method 1:

- When an item is deleted from the queue, move the entire queue to the **left** so that the first element is again at queue[0] and front is at **-1**. It should also recalculate **rear** so that it is correctly positioned.
- Shifting an array is very time-consuming when there are many elements in queue & queueFull has worst case complexity of **O(MAX_QUEUE_SIZE)**



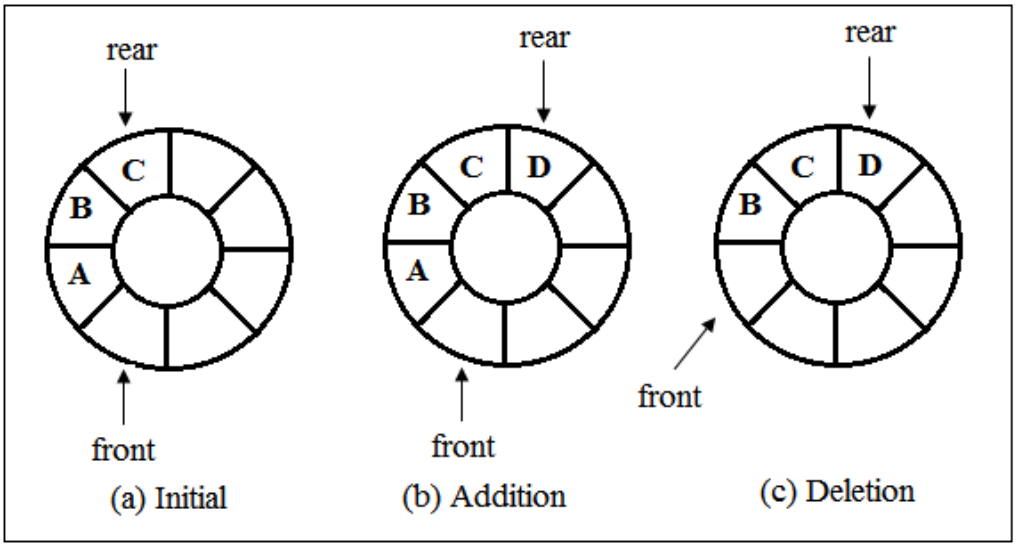
Method 2:

Circular Queue

- It is "The queue which **wrap around** the end of the array." The array positions are arranged in a circle.
- In this convention the variable **front** is changed. **front** variable points one position **counterclockwise** from the location of the front element in the queue. The convention for rear is unchanged.

CIRCULAR QUEUES

- It is "The queue which **wrap around** the end of the array." The array positions are arranged in a circle as shown in figure.
- In this convention the variable **front** is changed. **front** variable points one position **counterclockwise** from the location of the front element in the queue. The convention for rear is unchanged.



Implementation of Circular Queue Operations

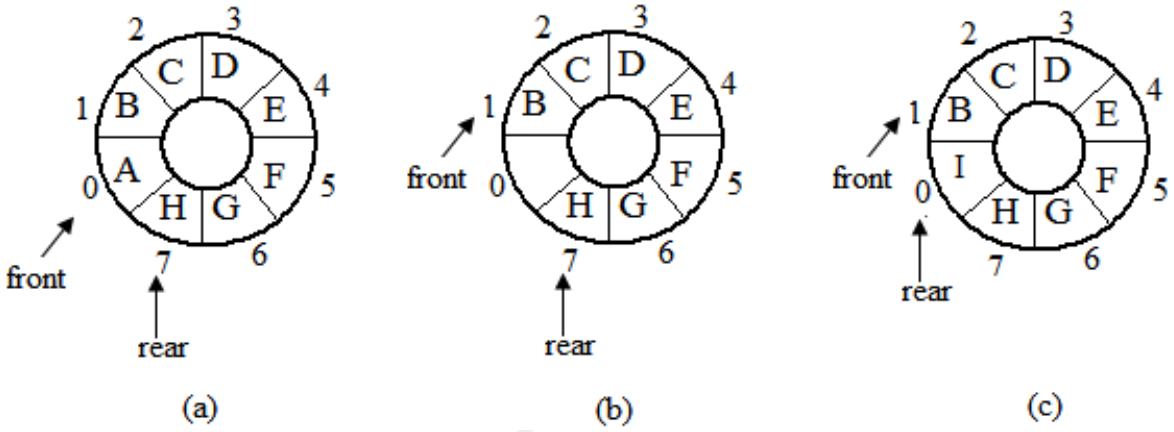
- When the array is viewed as a circle, each array position has a **next** and a **previous** position. The position next to **MAX-QUEUE-SIZE -1** is **0**, and the position that precedes **0** is **MAX-QUEUE-SIZE -1**.
- When the queue **rear** is at MAX_QUEUE_SIZE-1, the next element is inserted at position **0**.
- In circular queue, the variables **front** and **rear** are moved from their current position to the next position in clockwise direction. This may be done using code

```
if (rear = = MAX_QUEUE_SIZE-1)
        rear = 0;
else rear++;
```

Addition & Deletion

- To add an element, increment **rear** one position clockwise and insert at the new position. Here the **MAX_QUEUE_SIZE** is 8 and if all 8 elements are added into queue and that can be represented in below figure (a).
- To delete an element, increment **front** one position clockwise. The element **A** is deleted from queue and if we perform 6 deletions from the queue of Figure (b) in this fashion, then queue becomes empty and that **front =rear.**
- If the element **I** is added into the queue as in figure (c), then **rear** needs to increment by 1 and the value of rear is **8**. Since queue is circular, the next position should be 0 instead of 8.
  This can be done by using the modulus operator, which computes remainders.

(rear +1) % MAX_QUEUE_SIZE




```
void addq(element item)
        {               /* add an item to the queue */
                rear = (rear +1) % MAX_QUEUE_SIZE;
                if (front == rear)
                        queueFull();            /* print error and exit */
                queue [rear] = item;
        }
```

Program: Add to a circular queue

```
element deleteq()
        {               /* remove front element from the queue */
                element item;
                if (front == rear)
                        return queueEmpty( );           /* return an error key */
                front = (front+1)% MAX_QUEUE_SIZE;
                return queue[front];
}
```

Program: Delete from a circular queue

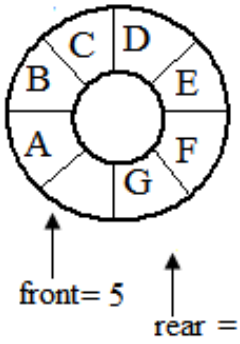
Note:

- When queue becomes empty, then **front =rear.** When the queue becomes full and **front =rear.** It is difficult to distinguish between an empty and a full queue.
- To avoid the resulting confusion, increase the capacity of a queue just before it becomes full.

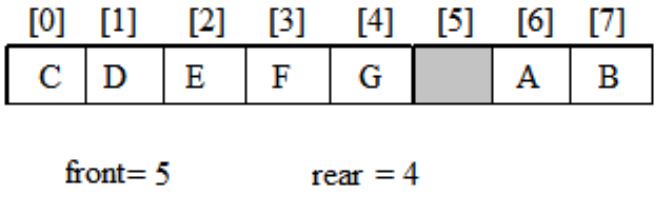
CIRCULAR QUEUES USING DYNAMIC ARRAYS

- A dynamically allocated array is used to hold the queue elements. Let **capacity** be the number of positions in the array queue.
- To add an element to a **full queue**, first increase the size of this array using a function **realloc**. As with dynamically allocated stacks, **array doubling** is used.

Consider the **full queue** of figure (a). This figure shows a queue with seven elements in an array whose capacity is 8. A circular queue is flatten out the array as in Figure (b).

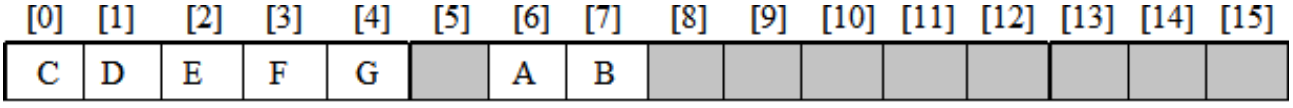


(a) A full circular queue



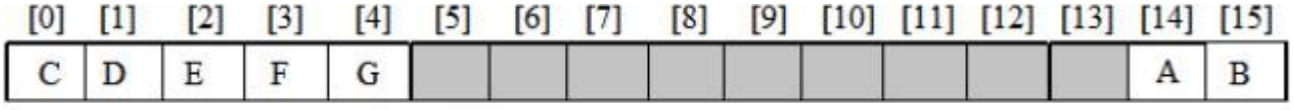
(b) Flattened view of circular full queue

Figure (c) shows the array after array doubling by relloc



(c) After array doubling

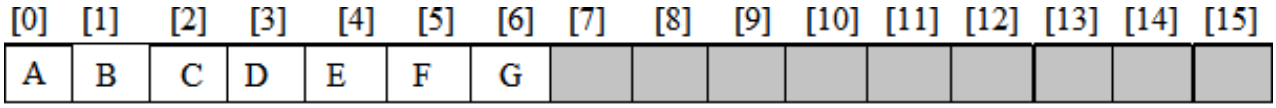
To get a proper circular queue configuration, slide the elements in the right segment (i.e., elements A and B) to the right end of the array as in figure (d)



(d After shifting right segment

To obtain the configuration as shown in figure (e), follow the steps

1) Create a new array **newQueue** of twice the capacity.
2) Copy the second segment (i.e., the elements queue [front +1] through queue [capacity-1]) to positions in **newQueue** beginning at 0.
3) Copy the first segment (i.e., the elements queue [0] through queue [rear]) to positions in newQueue beginning at **capacity – front – 1.**



front= 15 rear = 6

(e) Alternative configuration

Below program gives the code to add to a circular queue using a dynamically allocated array.

```
void addq( element item)
    {               /* add an item to the queue
        rear = (rear +1) % capacity;
        if(front == rear)
            queueFull( );          /* double capacity */
        queue[rear] = item;
    }
```

Below program obtains the configuration of **figure (e)** and gives the code for queueFull. The function copy (a,b,c) copies elements from locations **a** through **b-1** to locations beginning at **c**.

```
void queueFull( )
    {                              /* allocate an array with twice the capacity */
        element *newQueue;
        MALLOC ( newQueue, 2 * capacity * sizeof(* queue));
        /* copy from queue to newQueue */

        int start = ( front + ) % capacity;
        if ( start < 2)                /* no wrap around */
            copy( queue+start, queue+start+capacity-1,newQueue);
        else
        {                          /* queue wrap around */
            copy(queue, queue+capacity, newQueue);
            copy(queue, queue+rear+1, newQueue+capacity-start);
        }
```

```
        /* switch to newQueue*/
        front = 2*capacity – 1;
        rear = capacity – 2;
        capacity * =2;
        free(queue);
        queue= newQueue;
}
```

MULTIPLE STACKS AND QUEUES

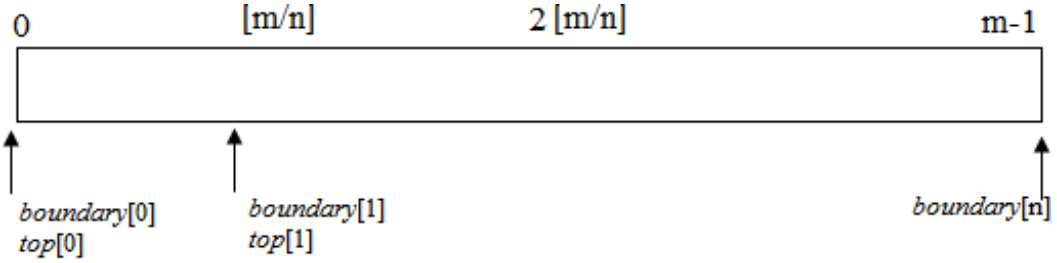
- In multiple stacks, we examine only **sequential mappings** of stacks into an array. The array is one dimensional which is **memory[MEMORY_SIZE]**. Assume ***n*** stacks are needed, and then divide the available memory into *n* segments. The array is divided in proportion if the expected sizes of the various stacks are known. Otherwise, divide the memory into equal segments.
- Assume that *i* refers to the stack number of one of the *n* stacks. To establish this stack, create indices for both the **bottom** and **top** positions of this stack. ***boundary*[i]** points to the position immediately to the left of the bottom element of stack *i*, ***top[i]*** points to the top element. Stack *i* is empty iff *boundary[i]=top[i]*.

The declarations are:

```
#define MEMORY_SIZE  100          /* size of memory */
#define MAX_STACKS  10            /* max number of stacks plus 1 */
element memory[MEMORY_SIZE];      /* global memory declaration */
int top [MAX_STACKS];
int boundary [MAX_STACKS] ;
int n;                            /*number of stacks entered by the user */
```

To divide the array into roughly equal segments

```
top[0] = boundary[0] = -1;
        for (j= 1;j<n; j++)
                top[j] = boundary[j]  = (MEMORY_SIZE / n) * j;
boundary[n]  = MEMORY_SIZE - 1;
```



All stacks are empty and divided into roughly equal segments

Figure: Initial configuration for *n* stacks in *memory [m]*.

In the figure, **n** is the number of stacks entered by the user, n < MAX_STACKS, and m =MEMORY_SIZE. Stack **i** grow from **boundary[i] + 1** to **boundary [i + 1]** before it is full. A Boundary for the last stack is needed, so set **boundary [n]** to **MEMORY_SIZE-1**.

Implementation of the add operation

```
void push(int i, element item)
    {           /* add an item to the ith stack */
        if (top[i] == boundary[i+l])
            stackFull(i);
        memory[++top[i]] = item;
    }
```

Program: Add an item to the ith stack

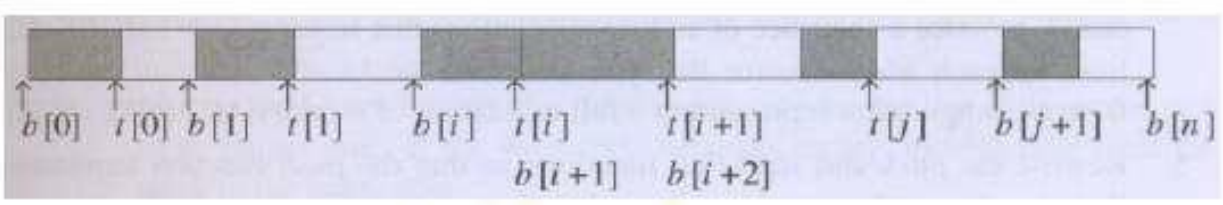
Implementation of the delete operation

```
element pop(int i)
    {           /* remove top element from the ith stack */
        if (top[i] == boundary[i])
            return stackEmpty(i);
        return memory[top[i]--];
    }
```

Program: Delete an item from the ith stack

The top[i] == boundary[i+1] condition in push implies only that a particular stack ran out of memory, not that the entire memory is full. But still there may be a lot of unused space between other stacks in array memory as shown in Figure.

Therefore, create an error recovery function called **stackFull** , which determines if there is any free space in memory. If there is space available, it should shift the stacks so that space is allocated to the full stack.



b boundary , t=top