UNIT-2

Prepared by:

Manjula L, Assistant Professor

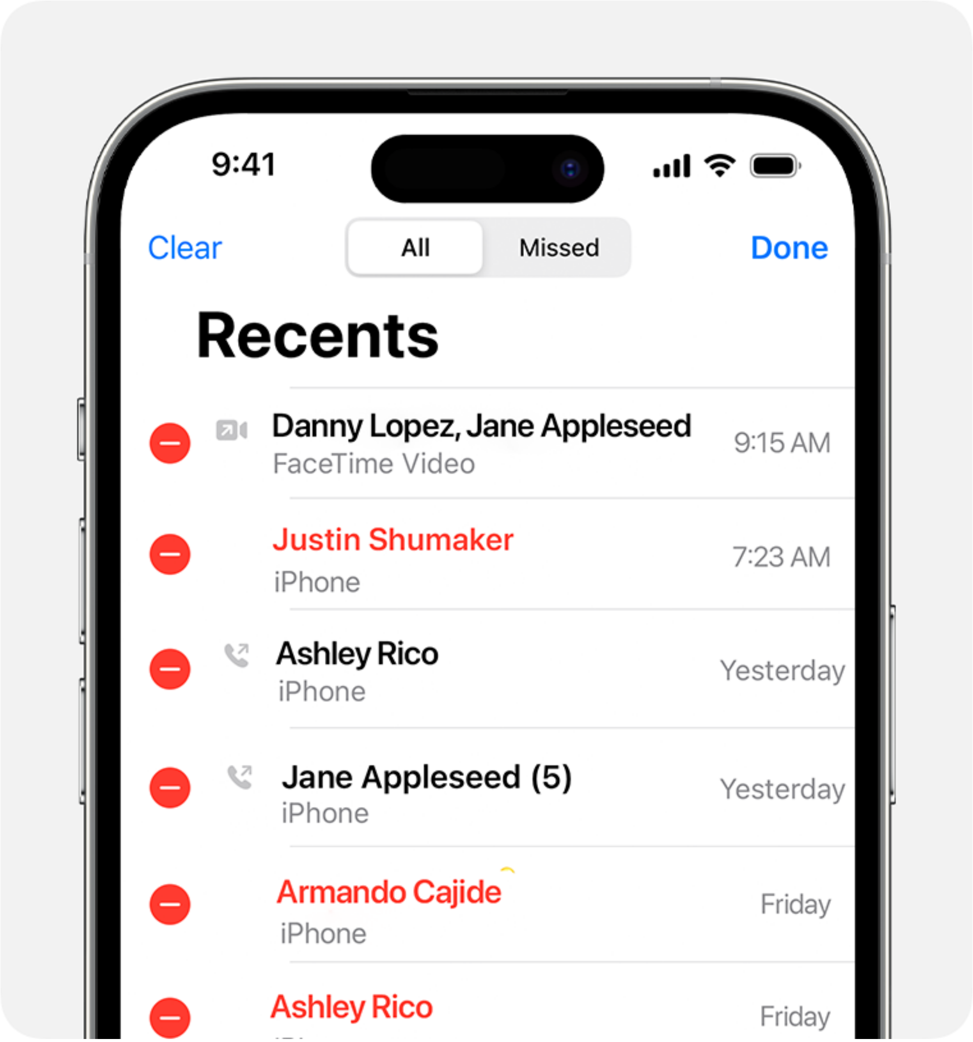
Dept. of CSE, MSRIT

Text Book: Horowitz, Sahni, Anderson-Freed: Fundamentals of Data Structures in C, 2nd Edition, Universities Press, 2008.

Contents

Stacks And Queues:

Simple Example

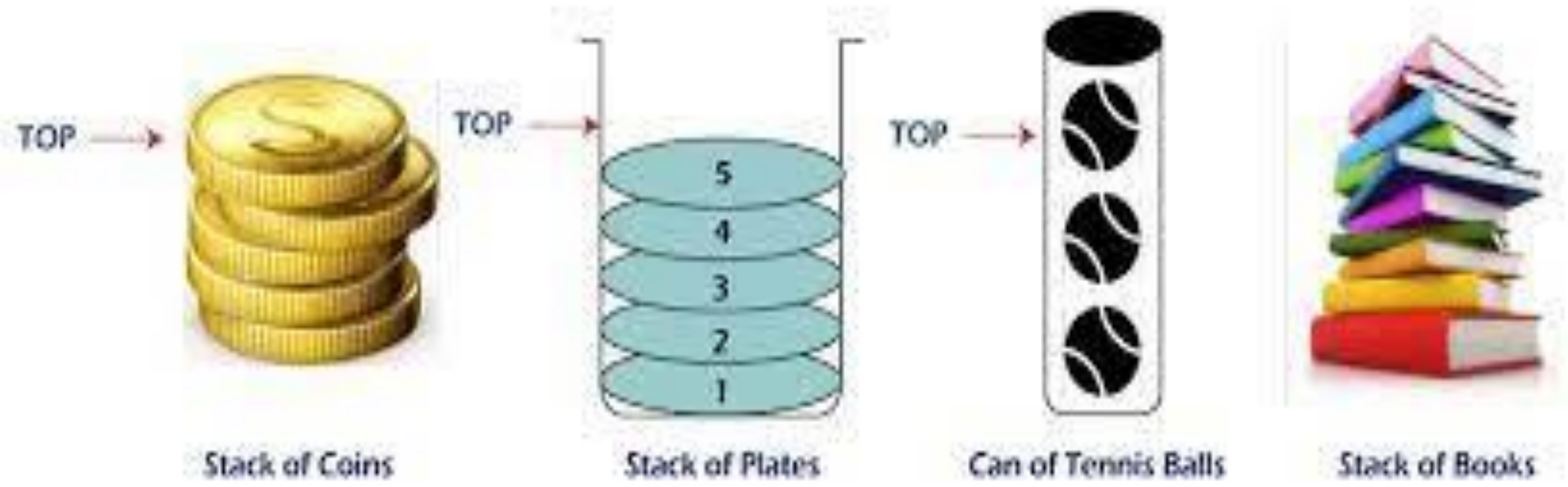


Call Logs in Phone – only Push / Insert

Stack

- *Stack is a ordered list in which insertions(push) and deletions (pop) are made at one end called **top**.*
- *Given a stack S = (*a_0*, … ….. * a_{n-1}*),* a_0 *is the bottom element* , a_{n-1} *is the top element.*

- *Stack always is LIFO (Last In First Out)*



An application of stack: stack frame of function call (activation record)

System stack is used by a program at runtime to process function calls. Whenever a function is invoked, the program creates a structure, referred to as an activation record or a stack frame, and places it on top of system stack.

fp: a pointer to current stack frame

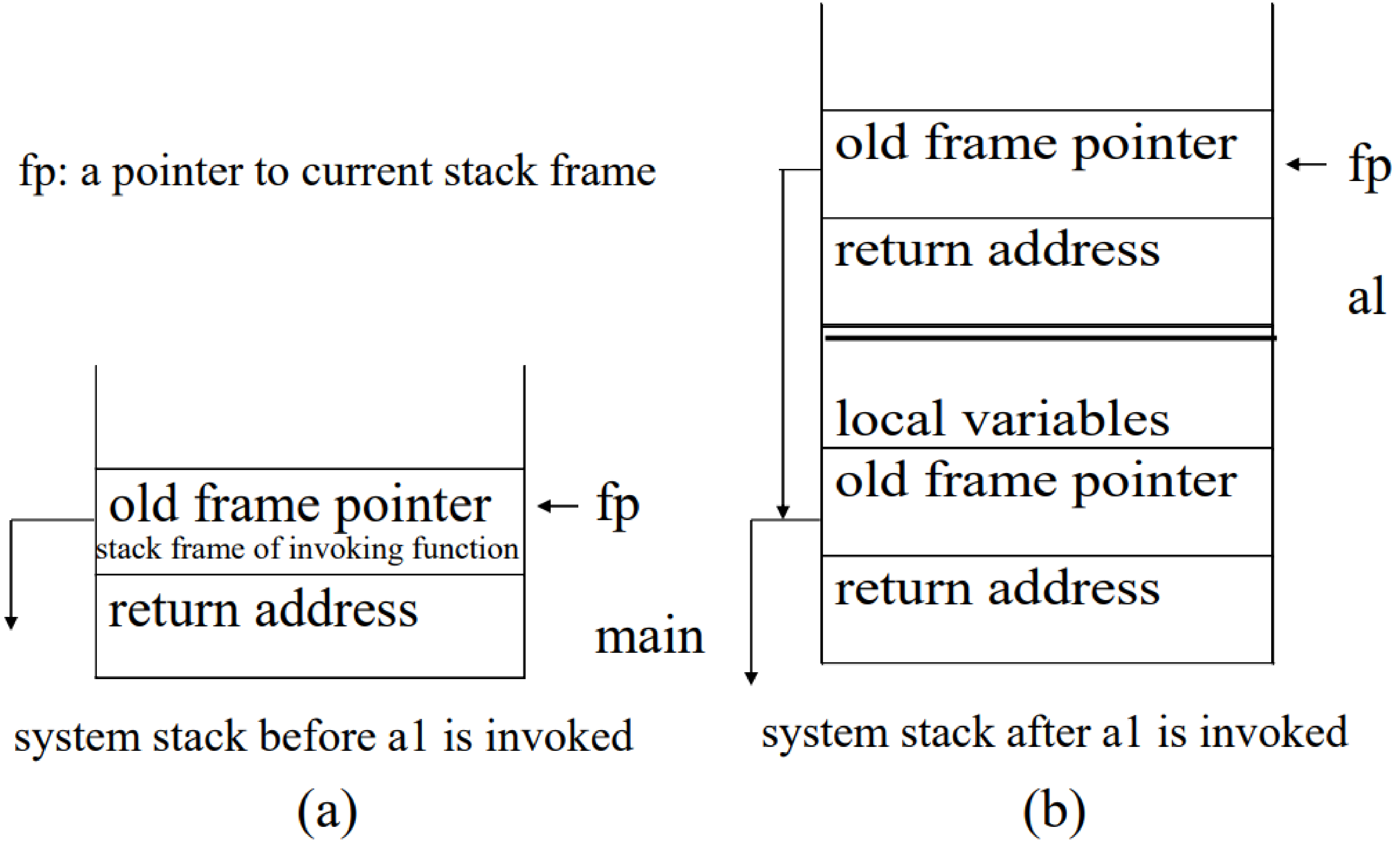


***Figure** System stack after function call

Stack: a Last-In-First-Out (LIFO) list

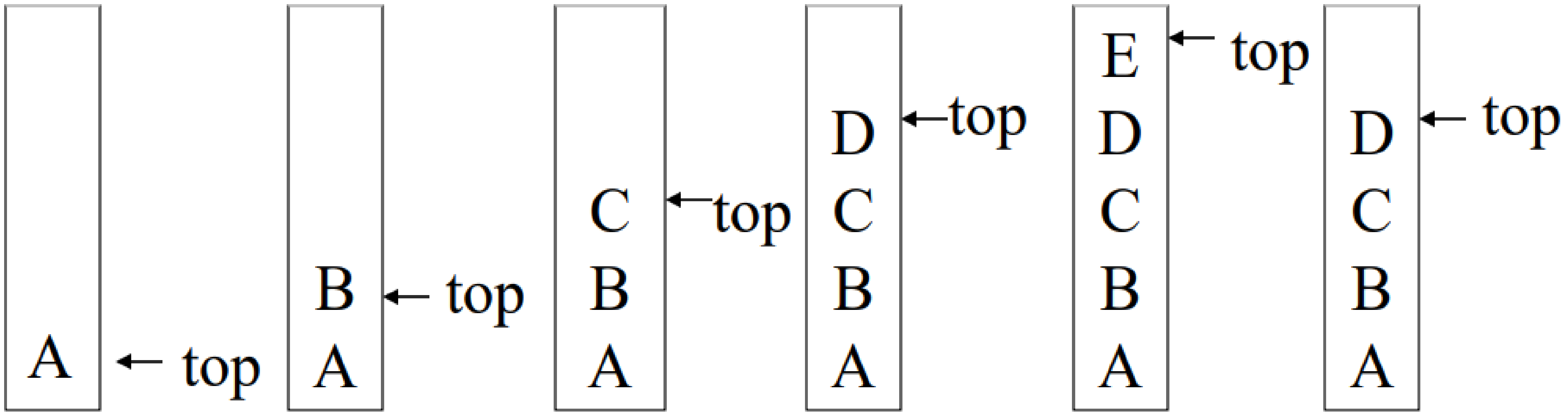


***Figure** Inserting and deleting elements in a stack

Abstract data type for stack

structure *Stack* is

objects: a finite ordered list with zero or more elements.

functions:

for all *stack* $\in$ *Stack*, *item* $\in$ *element*, *max_stack_size* $\in$ positive integer

Stack CreateS(*max_stack_size*) ::=
    create an empty stack whose maximum size is *max_stack_size*

Boolean IsFull(*stack, max_stack_size*) ::=
    if (number of elements in *stack* == *max_stack_size*)
    return TRUE
    else return FALSE

Stack Add(*stack, item*) ::=
    if (IsFull(*stack*)) *stack_full*
    else insert *item* into top of *stack* and **return**

Boolean IsEmpty(*stack*) ::=
if(*stack* == CreateS(*max_stack_size*))
return TRUE
else return FALSE
Element Delete(*stack*) ::=
if(IsEmpty(*stack*)) **return**
else remove and return the *item* on the top of the stack.

***Structure :** Abstract data type *Stack*

Implementation: using array

Stack **CreateS(max_stack_size)** ::=

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

Boolean **IsEmpty(Stack)** ::= top< 0;

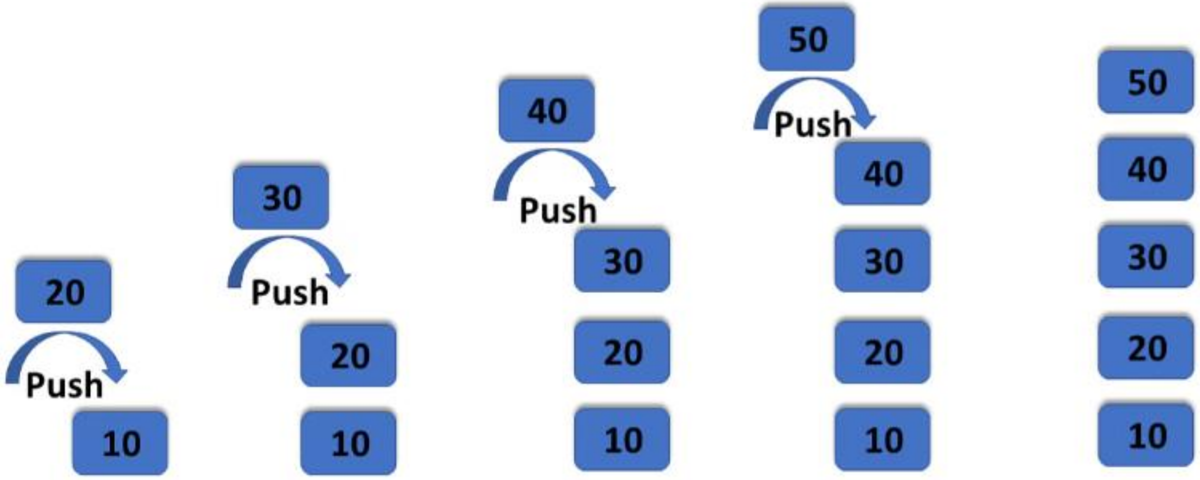
Boolean **IsFull(Stack)** ::= top >= MAX_STACK_SIZE-1;

Stack - function stack full

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

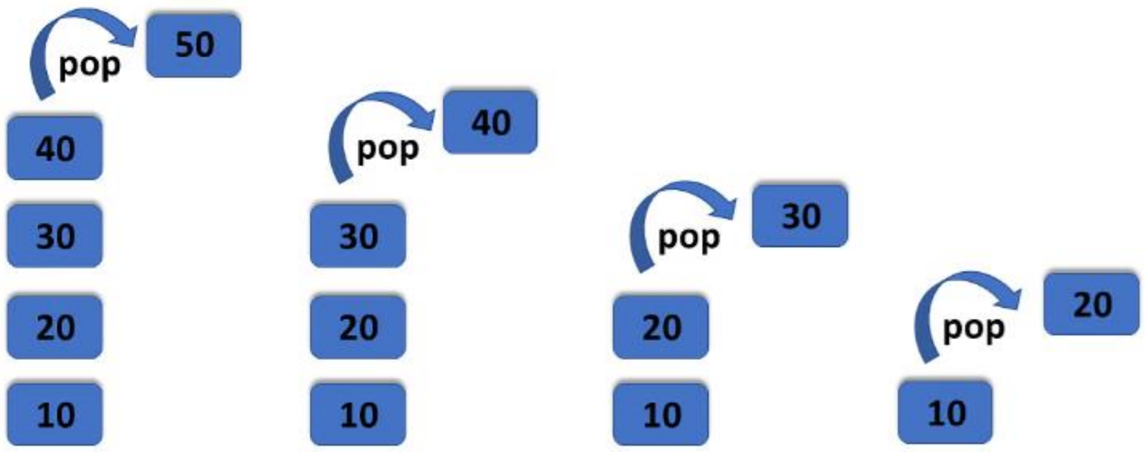
Stack – Push Function

```
void Push(element item )
{
        if (top >= MAX_STACK_SIZE  - 1)
                stackFull( );
        stack[++top] = item;
}
```



Stack – Pop Function

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



Stacks using Dynamic Arrays

```
Stack creates()::= typedef struct
                    {
                            int key;
                    }element;
                    element *stack;

          MALLOC(stack,sizeof(*stack));
                    int capacity=1;
                    int top=-1;
Boolean Isempty(Stack)::= top<0;
Boolean IsFull(Stack)::top>=capacity-1;
```

Stack Full with array doubling

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

Queue: a First-In-First-Out (FIFO) list

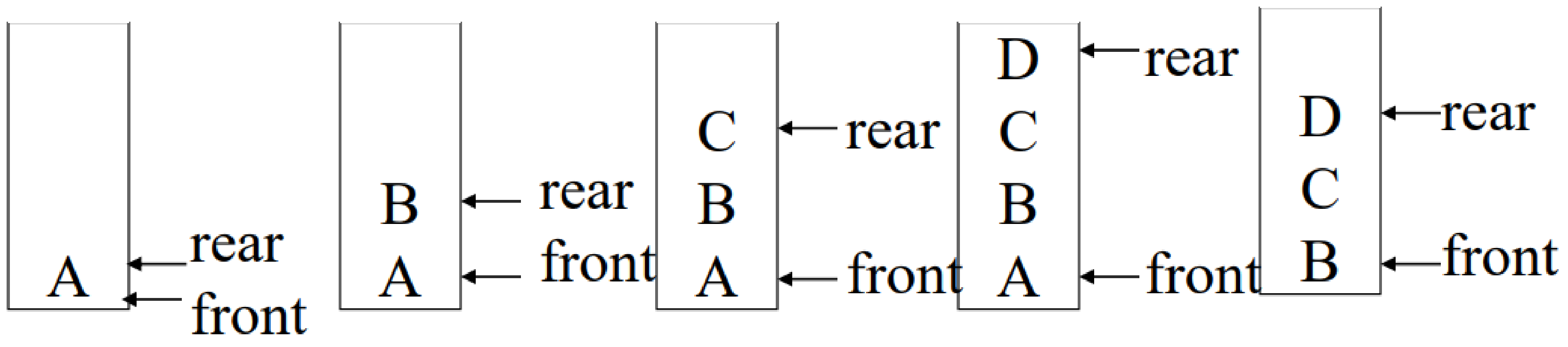


***Figure :** Inserting and deleting elements in a queue*

Application: Job scheduling

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

Abstract data type of queue

structure *Queue* is
objects: a finite ordered list with zero or more elements.
functions:
for all $queue \in Queue$, $item \in element$,
$max\_queue\_size \in$ positive integer
Queue CreateQ(*max_queue_size*) ::=
create an empty queue whose maximum size is
max_queue_size
Boolean IsFullQ(*queue, max_queue_size*) ::=
if(number of elements in *queue* == *max_queue_size*)
return *TRUE*
else return *FALSE*
Queue AddQ(*queue, item*) ::=
if (IsFullQ(*queue)) queue_full*
else insert *item* at rear of *queue* and return *queue*

```
Boolean IsEmptyQ(queue) ::=
        if (queue ==CreateQ(max_queue_size))
        return TRUE
        else return FALSE
Element DeleteQ(queue) ::=
        if (IsEmptyQ(queue)) return
        else remove and return the item at front of queue.
```

***Structure :** Abstract data type *Queue*

Implementation 1: using array

```
Queue CreateQ(max_queue_size) ::=
# 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;
Boolean IsEmpty(queue) ::= front == rear
Boolean IsFullQ(queue) ::= rear == MAX_QUEUE_SIZE-1
```

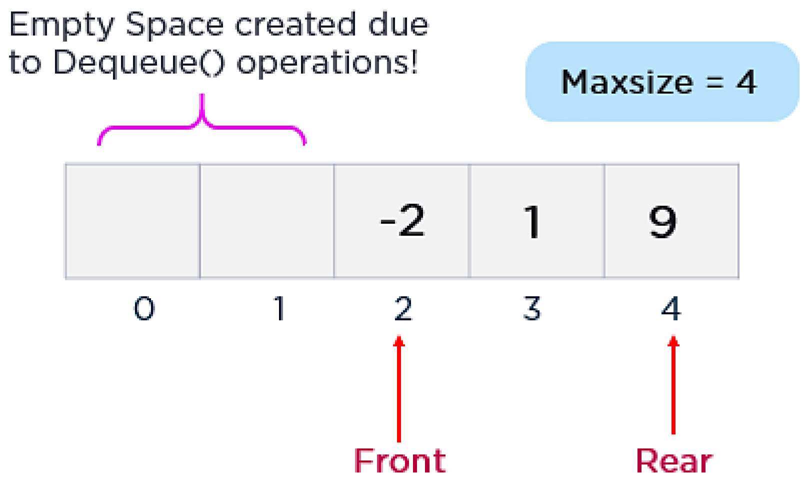
Queue – Add to Queue

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

Queue – Delete from Queue

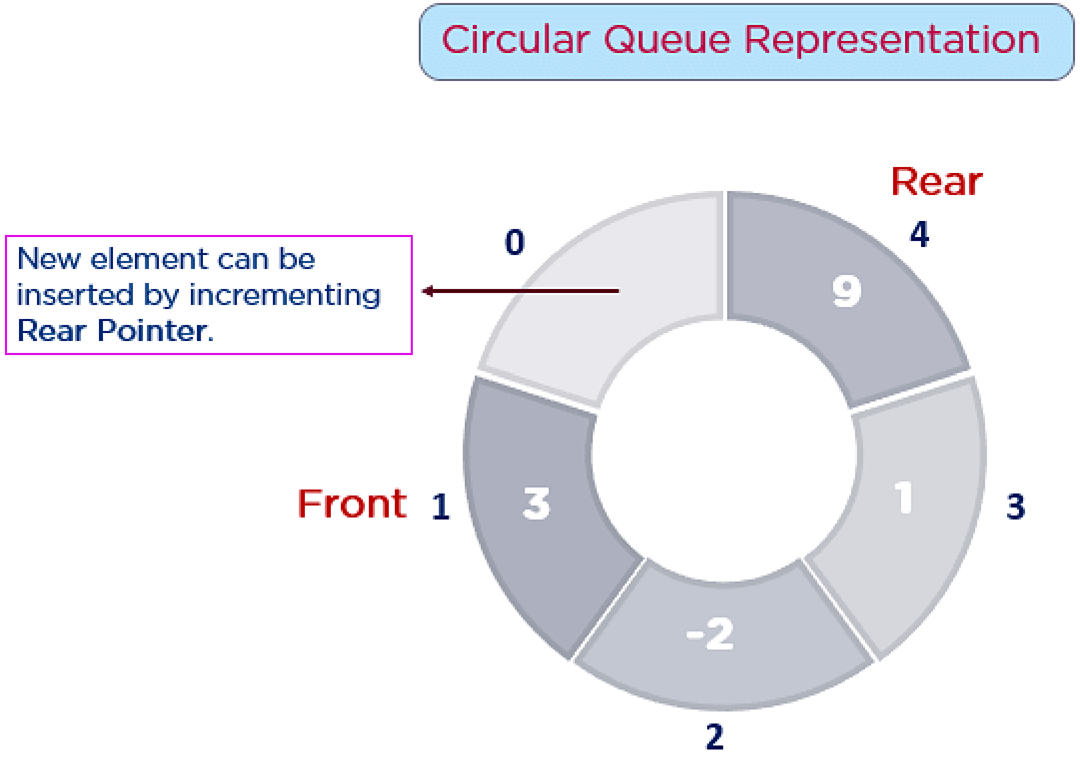
```
element deleteq( )
{   //remove element from the queue
        if (front == rear)
                return queueEmpty( );     // returns error key
        return queue[++front];
}
```

Circular Queue

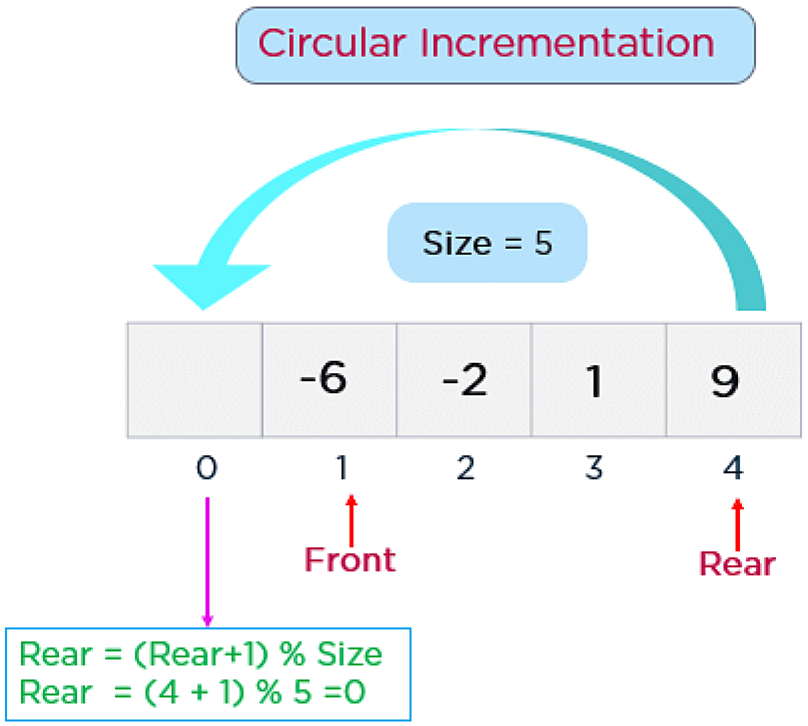


Implementation of a linear queue brings the drawback of memory wastage. However, memory is a crucial resource that you should always protect by analyzing all the implications while designing algorithms or solutions. In the case of a linear queue, when the rear pointer reaches the MaxSize of a queue, there might be a possibility that after a certain number of dequeue() operations, it will create an empty space at the start of a queue.

Circular Queue



A circular queue is an extended version of a linear queue as it follows the First In First Out principle with the exception that it connects the last node of a queue to its first by forming a circular link. Hence, it is also called a Ring Buffer.


Circular Incrementation
Size = 5
-6
-2
1
9
0
1
2
3
4
Front
Rear
Rear = (Rear+1) % Size
Rear = (4 + 1) % 5 =0

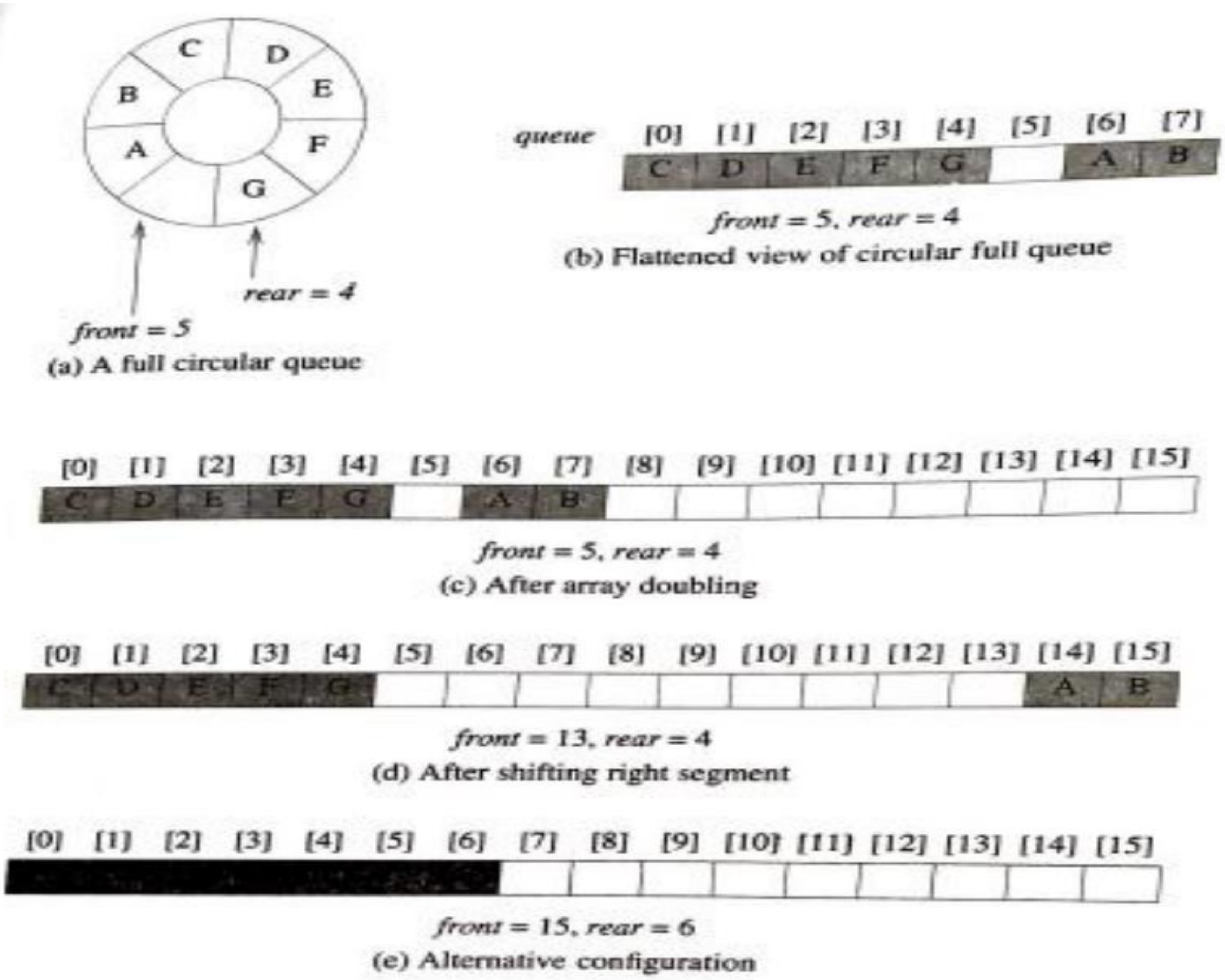
Circular Queue – Add

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

Circular Queue– Delete

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

Dynamic Circular Queue



(a) A full circular queue

(b) Flattened view of circular full queue

(c) After array doubling

(d) After shifting right segment

(e) Alternative configuration

Dynamic Circular Queue

C	D	E	F	G		A	B
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

1. Create a new array newQueue of twice the capacity.

Dynamic Circular Queue

				rear	front		
C	D	E	F	G		A	B
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

1. Create a new array newQueue of twice the capacity.

[0]	**[1]**	**[2]**	**[3]**	**[4]**	**[5]**	**[6]**	**[7]**	**[8]**	**[9]**	**[10]**	**[11]**	**[12]**	**[13]**	**[14]**	**[15]**

Dynamic Circular Queue

				rear	front		
C	D	E	F	G		A	B
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

2. Copy the second segment (i.e., the elements queue[front+1] through queue[capacity -1]) to positions in newQueue beginning at 0.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

Dynamic Circular Queue

				rear	front		
C	D	E	F	G		A	B
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

2. Copy the second segment (i.e., the elements queue[front+1] through queue[capacity -1]) to positions in newQueue beginning at 0.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
A															

Dynamic Circular Queue

				rear	front		
C	D	E	F	G		A	B
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

2. Copy the second segment (i.e., the elements queue[front+1] through queue[capacity -1]) to positions in newQueue beginning at 0.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
A	B														

Dynamic Circular Queue

				rear	front		
C	D	E	F	G		A	B
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

3. Copy the first segment (i.e., the elements queue[0] through queue[rear]) to positions in newQueue beginning at capacity-front-1.
   8-5-1= 3-1=2

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
A	B	C													

Dynamic Circular Queue

				rear	front		
C	D	E	F	G		A	B
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

3. Copy the first segment (i.e., the elements queue[0] through queue[rear]) to positions in newQueue beginning at capacity-front-1.
   8-5-1= 3-1=2

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
A	B	C	D	E	F	G									

Dynamic Circular Queue

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
A	B	C	D	E	F	G									
															front

front = 2 * capacity - 1;

Dynamic Circular Queue

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
A	B	C	D	E	F	G									
						rear									front

front = 2 * capacity - 1;

rear = capacity-2; =8-2 =6

Capacity*=2

Doubling Queue Capacity

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.

   start = front+1 % capacity =(5+1)%8 = 6

   Copy(queue+start, queue+capacity, newQueue)
3. Copy the first segment (i.e., the elements queue[0] through queue[rear]) to positions in newQueue beginning at capacity-front-1.

   Copy(queue, queue+rear+1, newQueue+capacity-start)

	A	B	C	D	E	F	G
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
FRONT							REAR

```
Start = (front+1)  % capacity
          1   %  8 =1
If (start < 2)
        copy( queue+start, queue+start+capacity-1 , newQueue);
```

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

Evaluation of Expressions

1. Expressions
2. Evaluating Postfix Expressions
3. Infix to Postfix Expressions

Expressions

X = a / b - c + d * e - a * c

a = 4, b = c = 2, d = e = 3

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

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

How to generate the machine instructions corresponding to a given expression? precedence rule + associative rule

Token	Operator	Precedence[1]	Associativity
( ) [ ] -> .	function call array element struct or union member	17	left-to-right
-- ++	increment, decrement[2]	16	left-to-right
-- ++ ! - - + & * sizeof	decrement, increment[3] 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
* / %	mutiplicative	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

user **compiler**

Infix	Postfix
2+3*4	234*+
a*b+5	ab*5+
(1+2)*7	12+7*
a*b/c	ab*c/
(a/(b-c+d))*(e-a)*c	abc-d+/ea-*c*
a/b-c+d*e-a*c	ab/c-de*ac*-

Token	Stack [0]	[1]	[2]	Top
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
-	6/2-3			0
4	6/2-3	4		1
2	6/2-3	4	2	2
*	6/2-3	4*2		1
+	6/2-3+4*2			0

***Figure 3.14:** Postfix evaluation of expn : 6 2 / 3 – 4 2 * +

```
int eval(void)
{/* evaluate a postfix expression, expr, maintained as a
    global variable. '\0' is the the end of the expression.
    The stack and top of the stack are global variables.
    getToken is used to return the token type and
    the character symbol. Operands are assumed to be single
    character digits */
  precedence token;
  char symbol;
  int op1, op2;
  int n = 0; /* counter for the expression string */
  int top = -1;
  token = getToken(&symbol, &n);
  while (token != eos) {
    if (token == operand)
      push(symbol-'0'); /* stack insert */
    else {
      /* pop two operands, perform operation, and
         push result to the stack */
      op2 = pop(); /* stack delete */
      op1 = pop();
      switch(token) {
        case plus: push(op1+op2);
                   break;
        case minus: push(op1-op2);
                    break;
        case times: push(op1*op2);
                    break;
        case divide: push(op1/op2);
                     break;
        case mod: push(op1%op2);
      }
    }
    token = getToken(&symbol, &n);
  }
  return pop(); /* return result */
}
```

```
int eval(void)
{
        precedence token;
        char symbol;
        int op1, op2;
        int n = 0; /* counter for the expression string */
        int top = -1;
        token = get_token(&symbol, &n);
        while (token != eos) {
        if (token == operand)
                push(symbol-'0'); /* stack insert */
```

```
else {
        op2 = pop(); /* stack delete */
        op1 = pop();
        switch(token) {
            case plus: push( op1+op2); break;
            case minus: push( op1-op2); break;
            case times: push( op1*op2); break;
            case divide: push( op1/op2); break;
            case mod: push(op1%op2);
        }
    }
    token = get_token (&symbol, &n);
    }
    return pop(); /* return result */
}
```

```
precedence get_token(char *symbol, int *n)
{
        *symbol =expr[(*n)++];
        switch (*symbol) {
                case ‘(‘ : return lparen;
                case ’)’ : return rparen;
                case ‘+’: return plus;
                case ‘-’ : return minus;
                case ‘/’ : return divide;
                case ‘*’ : return times;
                case ‘%’ : return mod;
                case ‘\0‘ : return eos;
                default : return operand;
                }
}
```

Conversion of infix to postfix

Token	Stack			Top	Output
	[0]	[1]	[2]		
a				-1	a
$*_1$	$*_1$			0	a
(	$*_1$	(		1	a
b	$*_1$	(		1	ab
+	$*_1$	(	+	2	ab
c	$*_1$	(	+	2	abc
)	$*_1$	match)		0	abc+
$*_2$	$*_2$	$*_1 = *_2$		0	$abc+*_1$
d	$*_2$			0	$abc+*_1d$
eos	$*_2$			0	$abc+*_1d*_2$

Conversion of infix to postfix

Rules

(1) Operators are taken out of the stack as long as their in-stack precedence is higher than or equal to the incoming precedence of the new operator.

(2) (has low in-stack precedence, and high incoming precedence.

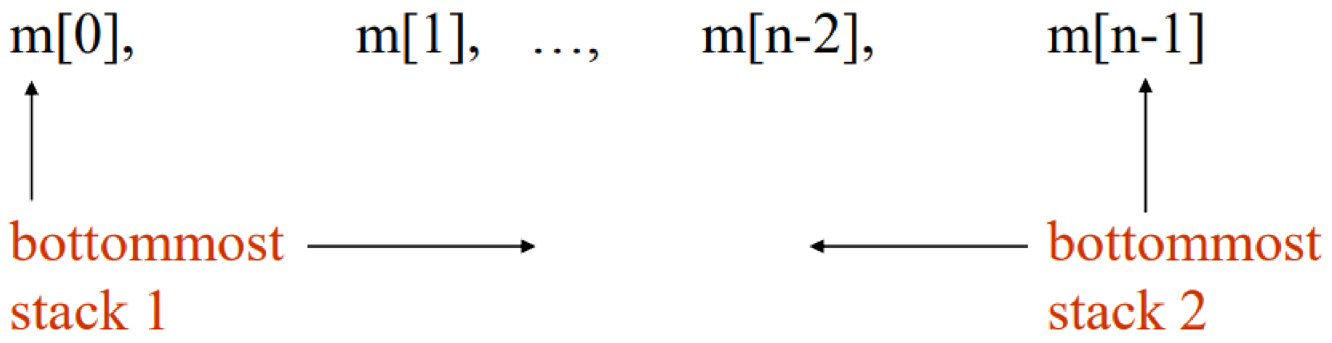
	(	)	+	-	*	/	%	eos
isp	0	19	12	12	13	13	13	0
icp	20	19	12	12	13	13	13	0

```
oid postfix(void)

    char symbol;
    precedence token;
    int n = 0;
    int top = 0; /* place eos on stack */
    stack[0] = eos;
    for (token = get _token(&symbol, &n); token != eos;token = get_token(&symbol,
    {
        if (token == operand)
        printf (“%c”, symbol);
        else if (token == rparen )
        {
```

```
/*unstack tokens until left parenthesis */
while (stack[top] != lparen)
            print_token(delete(&top));
delete(&top); /*discard the left parenthesis */
}
else{
                  /* remove and print symbols whose isp is greater than or equal to the
                                          current token’s icp */
      while(isp[stack[top]] >= icp[token] )
            print_token(delete(&top));
      add(&top, token);
      }
}
while ((token = delete(&top)) != eos)
print_token(token);
print(“\n”);
}
```

Multiple Stacks and Queue



If two stack to be implemented for a given memory. The first stack grows forward and other stack grows from the last element (backward)

More than two stacks (n) memory is divided into n equal segments

boundary[stack_no]

$0 \leq$ stack_no < MAX_STACKS

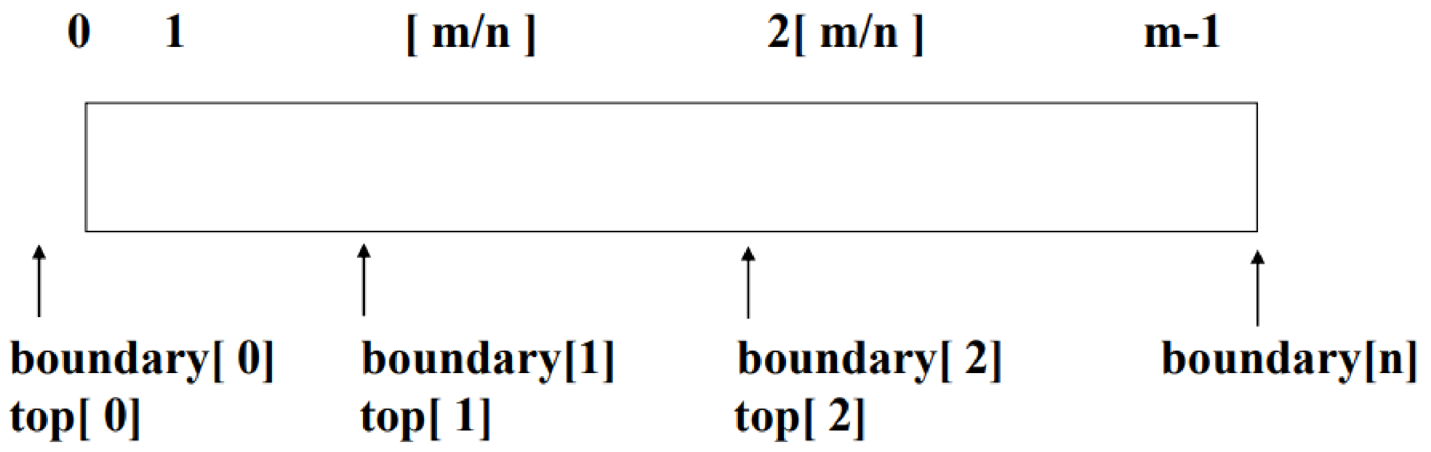
top[stack_no]

$0 \leq$ stack_no < MAX_STACKS

m =size

n= Number of Divisions

Initially, boundary[i]=top[i].



All stacks are empty and divided into roughly equal segments.

```
#define MEMORY_SIZE 100                    /* size of memory */
#define MAX_STACK_SIZE 100
                                /* max number of stacks plus 1 */
element memory[MEMORY_SIZE];
int top[MAX_STACKS];
int boundary[MAX_STACKS];
int n;                    /* number of stacks entered by the user*/
top[0] = boundary[0] = -1;
for (i = 1; i < n; i++)
        top[i] =boundary[i] =(MEMORY_SIZE/n)*i;
boundary[n] = MEMORY_SIZE-1;
```

```
void add(int i, element item)
{
  /* add an item to the ith stack */
  if (top[i] == boundary [i+1])
     stack_full(i);     may have unused storage
     memory[++top[i]] = item;
}
```

***Program 3.12:**Add an item to the stack *stack-no* (p.129)

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

***Program 3.13:**Delete an *item* from the stack *stack-no* (p.130)

StackFull in Multiple stack Implementation

- There are several ways that we can design stackFull so that we can add elements to this stack until the array is full. We need to guarantee that stackFull adds elements as long as there is free space in array memory.

Find j, stack_no < j < n

such that top[j] < boundary[j+1]

or, 0 ≤ j < stack_no

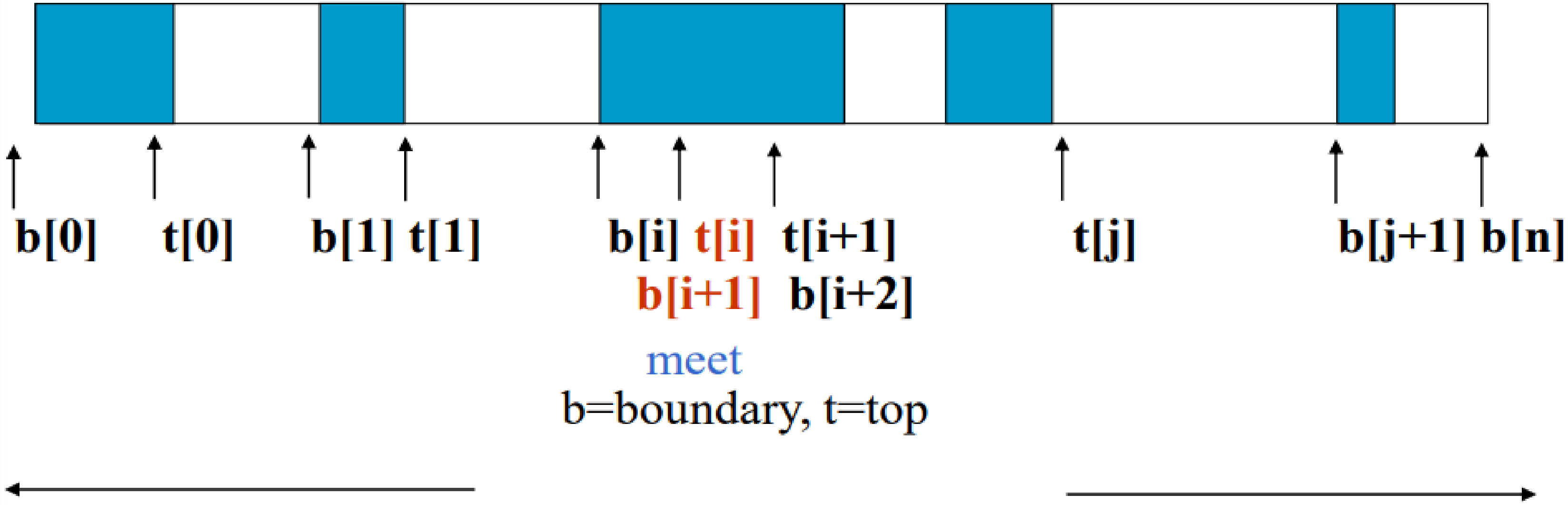


***Figure 3.19:** Configuration when stack i meets stack i+1, but the memory is not full (p.130)