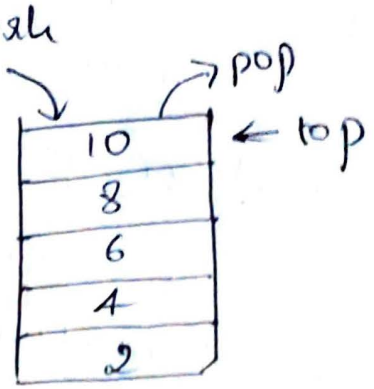
MODULE-2

STACKS AND QUEUE

Definition:→ Stack is a non-primitive linear data structure which additions of new elements & deletion of data element are done at one end called top of the stack.

Inserting and deletion of elements takes place at one end the last element added to the stack will be the first item to be deleted from the stack. Hence the stack is called Last In first out & First In Last out (FILO & LIFO) data structure.

Example:-



The various operations that can be followed on stacks are

* Insert an item into the stack (PUSH)
* Delete an item from the stack (pop)
* Display the contents of the stack (Display)

Consider a stack of four elements 30, 15, 20 and 10

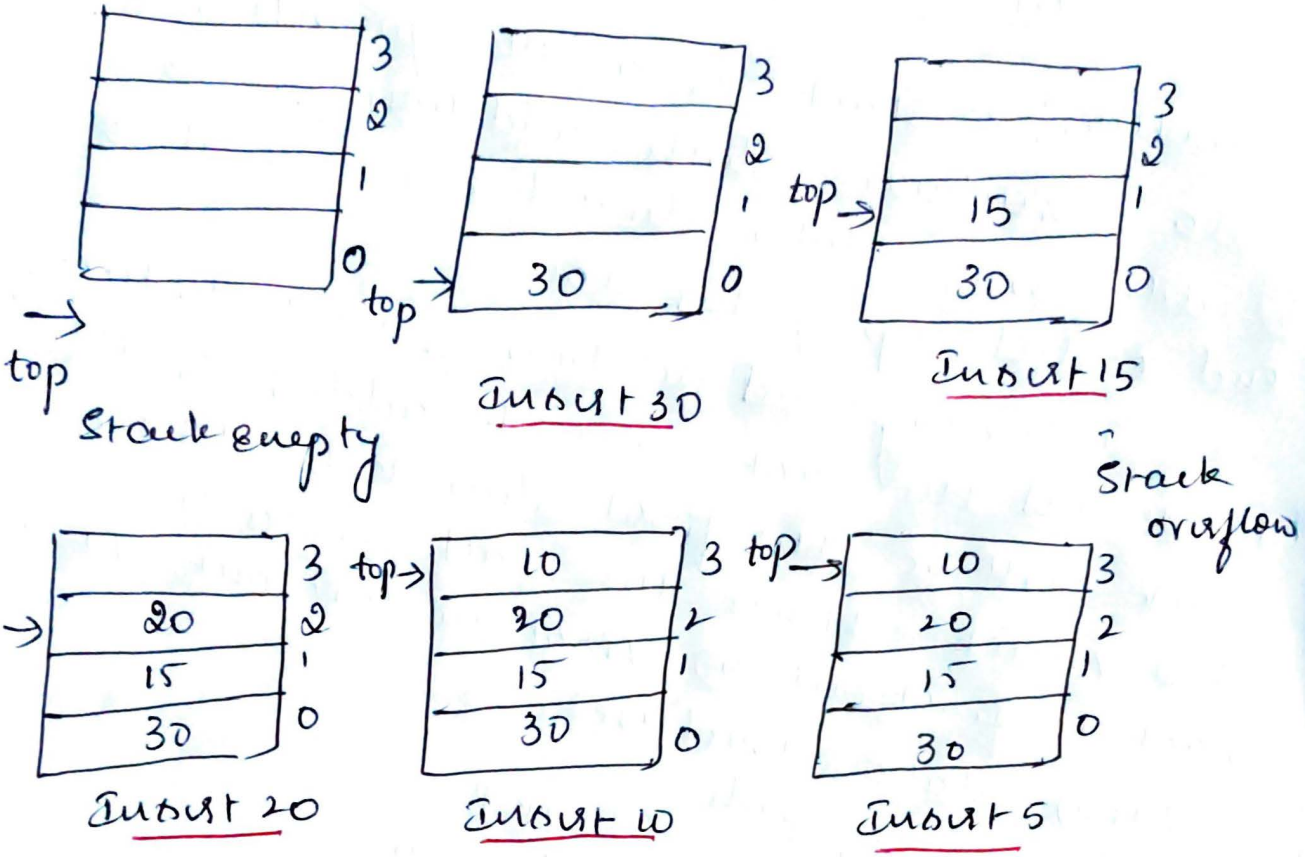


fig:- Insertion operation

Initially stack is empty and top points to -1. As the items are inserted onto the stack the top is incremented by 1. for the above example after inserting 10 the stack is full. It is not possible to insert a new item. This situation is called stack overflow.

Initially top = -1.

Insert 30 top = -1 + 1 = 0

as 0th position 30 is inserted

Insert 15 top = 0 + 1 = 1

at 1st position 15 is inserted

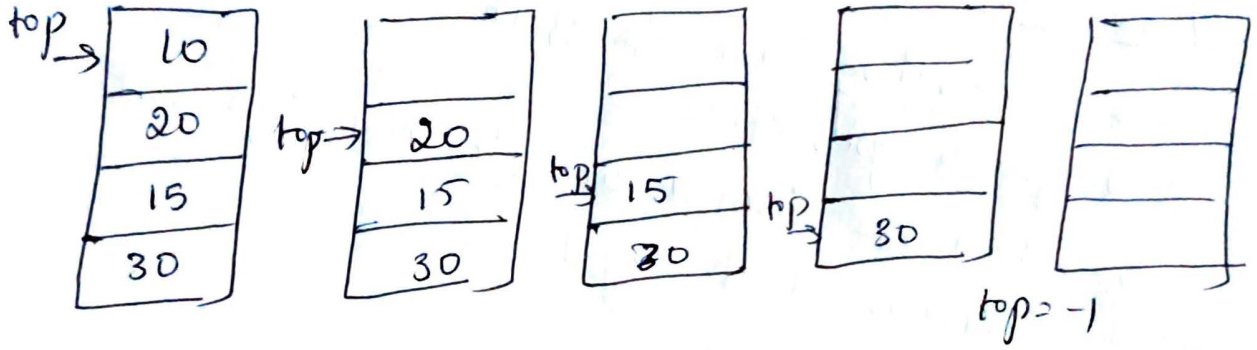
After inserting 10

top = 3

Insert 5

top = 3 + 1 = 4

Stack overflow condition occurs. Where an item is to be deleted it should be deleted from the top as in fig.



As the items are deleted from the stack top is decremented by 1. for the above example after deleting 30 Stack is Empty. This situation is called stack Empty & stack overflow

Representing Stack in C:

1) The Stack can be represented as an array to store similar type of elements in continuous order and top is used to indicate the position and is used as index of the array. An stack consists of two elements

* an array
* top

b) Stack can also be represented as a structure having two members

→ Items is an array of max element which is used to store elements of stack

→ top is used to indicate the current position of the stack

* push/insert:-

→ Inserting an element into a stack is called push operation

→ when we insert an element top value will be incremented by 1

i.e top = top + 1

Algorithm:-

1) Accept the structure stack along with the new element which is to be inserted on to stack

2) check whether stack is full or not-

if top = MAXSIZE - 1

3) If the stack is not full increment the top by 1 and insert the new element.

4) If the stack is full print the message as stack full.

Implementation of push operation:

```
void push()
{
    int x;
    if (top == MAXSIZE-1)
```

```
{
printf("Stack is full");
return;
}
printf("Enter element in");
scanf("%d", &x);
top=top+1; /* Increment top by 1 */
Stack[top]=x; /* Insert the element */
}
```

Pop operation;-

→ Deleting an element from the stack is called as pop operation

→ when we delete an element top is decremented by 1

Algorithm;-

1) Accept the structure stack as an argument

2) check whether stack is Empty @ not

* If the stack is Empty. print the message stack is ~~overflow~~ underflow

* If the stack is not Empty. decrement the top value by 1

Implementation of pop:-

```
void pop()
{
if (top==-1)
{
printf ("stack is empty /underflow\n");
return;
}
printf ("popped element is %d",
                stack[top]);
  top= top-1; /* deleting element from
                  the stack */
}
```

Display:->

In the display procedure.

-> If the stack has some elements. all the elements are displayed one after the other

-> If there are no items present the appropriate error message is displayed

ADT:- 1) stack create (maxstacksize)
2) Boolean IsFull (stack,)
3) push (stack, elem)
4) Boolean IsEmpty (stack)
5) pop (stack)

Implementation of display operation

```
void display()
{
 int i;
printf("\n Stack status \n");
 if(top==-1)
{
printf("Stack is empty \n");
return;
}
else
 for(i=top;i>=0;i--)
 printf("%d \n", stack[i]);
}
```

Write a program to construct stack of integers and perform the following operations on it (using arrays)

a) push b) pop c) display

```
#include<stdio.h>
#define MAXSIZE 4
 int stack[MAXSIZE], top=-1;
 void main()
 {
  int ch, n,i;
  void push();
  void pop();
  void display();
```

```
while(1)
{
clrscr();
 printf("1. push \n");
 printf("2. pop \n");
 printf("3. display \n");
printf("4. Exit \n");
printf("Enter the choice \n");
 scanf("%d", &choice);
 switch (choice)
 {
  case 1: push();
          break;
  case 2: pop();
          break;
  case 3: display();
          break;
  case 4: return;
  } /* switch */
 } /* while */
 getch();
} * main */

void push()
  {
    int x;
    if (top==MAXSIZE-1)
```

```
{
printf("stack overflow\n");
return;
}
printf("\n enter element- ");
scanf("%d", &x);
top= top+1;
stack[top] = x;
}
void pop()
{
if (top==-1)
{
printf("stack is empty\n");
return;
}
printf("poped element is %d", stack[top]);
top= top-1;
}
void display()
{
int i;
printf("\n stack status\n");
if (top==-1)
{
printf("stack empty\n");
return;
}
```

```
for(i=0;i<=top;i++) / for(i=top;i>=0;i--)
    printf("\n %d", stack[i]);
}
```

```
void display()
{
int i;
printf ("\n Stack Status");
if (top==-1)
{
printf ("Stack Empty \n");
return;
}
for(i=0; i<=top; i++)
    Printf ("\n%d", stack[i]);
}
```

Applications of Stack

The various applications in which stack are used:

1. **Conversion of expressions:**

When we write mathematical expression in a program, we use infix expressions. These expressions will be converted into equivalent machine instructions by the compiler using stacks. Using stacks we can efficiently convert the expressions from infix to postfix, infix to prefix, postfix to infix, postfix to prefix, postfix to infix and postfix to prefix.

2. **Evaluation of Expression:**

An arithmetic expression represented in the form of either postfix or prefix can be easily evaluated.

3. **Recursion:**

A function which calls itself is called recursive function. Some of the problems such as towers of Hanoi,

problems involving tree manipulations etc. can be implemented very efficiently using recursion. It is a very important facility available in a variety of programming lang such as C, C++ etc.

4. Other Applications:

There are so many other appl. where stacks can be used.

- to find String is palindrome
- to check whether a given expr. is valid or not.
- topological sort.

Conversion of Expressions

An arithmetic expression is successively defined as follows:

1. if a is an operand (can be variable or numeric constant) then a is an arithmetic expression.
2. If a and b are arithmetic expns then

   a op b

   is an arithmetic expr. where op is any operator such as +, −, *, / and so on.

In postfix expr, there is no need of operator precedence and other rules.

In postfix expression during scanning the topmost operands are popped off and are calculated applying the encountered operator.

An arithmetic expression can be classified into three categories: 7

1. Infix expression
2. Postfix (Suffix) expression
3. Prefix expression

Infix Expression

Arithmetic expression in which an operator is in between two operands is called an infix expression. The infix expression can be parenthesized or un-parenthesized.

Ex: a+b

a*b/2

(a+b+c)/2

((3+2)*5)/4

The first two expressions are ~~except~~ un-parenthesized expr where as the last two expressions are parenthesized exprs.

Postfix Expression

Arithmetic expression in which operator is placed after operands is called postfix expression.

Ex: ab+

ab*/2

ab+c+2/

32+5*4/

postfix expressions does not have any parenthesis.

Adv :- In ~~post~~ Infix exp set of rules must be applied to expr inorder to determine the final value

← In Infix if operator occurs, precedence must be applied to determine which operator to be solved first.

Prefix Expression

Arithmetic expression in which operator is placed before the operands is called prefix expression.

Ex:- +ab

*ab/2

/++abc2

/*+3254

Like postfix expressions, prefix expr. do not contain any parentheses.

Conversion from infix to postfix

① Convert A+B*C to postfix expr.

A+B*C

Here * is having higher precedence than +, so, B*C is evaluated first. postfix form of B*C is BC*. Now, the expr. is A+BC*

Next + is evaluated

ABC*+

Note:-

Operator	Precedence
()	first
Exponentiation	Second
Mul/Div	Third
Add/Sub	Fourth

A + (B * C)
A + (BC *)
ABC * +

operator called

(2) Convert (A+B) * C to postfix.

(A+B) * C
(AB+) * C
(AB+) C *
AB+C *

- expr within parenthesis is evaluated first AB+
- Next * is evaluated

ix expr.

(3) A + (B * C)

= A + (BC *)
= ABC * +

(4) (A+B) / (C - D)

= (AB+) / (C - D)
= (AB+) / (CD-)
= (AB+) (CD-) /
= AB+ CD- /

(5) (A - B) * (D / E)

= (AB-) * (DE/)
= (AB-) (DE/) *
= AB- DE/ *

odence

lt.

ow,

Need of converting infix to postfix: - In postfix expression there are no parenthesis and no precedence rules to learn. Because of this simplicity, handheld calculators use postfix notation to avoid complications of parentheses

⑥ $(A+B \uparrow D) / (E-F) + G$

$(A+(B D \uparrow)) / (E-F)+G$

$(A+(B D \uparrow)) / (E F-)+G$

$(A B D \uparrow+) / (E F-)+G$

$((A B D \uparrow+)(E F-)/)+G$

$A B D \uparrow+ E F-/ G+$

⑦ $A * (B+D) / E - F * (G+H/K)$

$A * (B D+) / E - F * (G+(H K/))$

$A * (B D+) / E - F * (G H K/+)$

$(A (B D+) *) / E - F * (G H K/+)$

$((A B D+ *) E /) - F * (G H K/+)$

$(A B D+ * E /) - ((A) (G H K/+) *)$

$(A B D+ * E /) (F G H K/+ *) -$

$A B D+ * E / F G H K/+ * -$

Consider the following arithmetic expr (postfix) (a) Convert it to infix expr. 9

12, 7, 3, −, /, 2, 1, 5, +, *, +

Scanning from left to right, translate each operator from infix postfix to infix. Use brackets to denote partial translation

~~= 12, (7 3 −) / 2 1 5 + * +~~

~~= (12 / (7 3 −)) 2 1 5 + * +~~

~~= (12 / (7 3 −)) 2 (1 + 5) * +~~

~~= (12 / (7 3 −)) 2 (1 +~~

= 12, 7, 3, −, /, 2, 1, 5, +, *, +

= 12 (7 − 3) / 2 1 5 + * +

= (12 / (7 − 3)) 2 1 5 + * +

= (12 / (7 − 3)) 2 (1 + 5) * +

= (12 / (7 − 3)) (2 * (1 + 5)) +

= (12 / (7 − 3)) + (2 * (1 + 5))

= 12 / (7 − 3) + 2 * (1 + 5)

(b) Evaluate the above expr.

= 12/4 + 2 * 6

= 3 + 12 = 15

12, 7, 3, −, /

Convert the foll. to infix expr.

① ABD↑ + EF − / G +

AB BD

A (B ↑ D) + E F − / G +

(A + (B ↑ D)) E F − / G +

(A + (B ↑ D)) (E − F) / G +

((A + (B ↑ D)) / (E − F)) G +

(A + (B ↑ D)) / (E − F) + G

)(A + B ↑ D) / (E − F) + G

② AB − DE / *

(A − B) DE / *

(A − B) (D / E) *

(A − B) * (D / E)

③ AB + CD − /

(A + B) (C − D) /

(A + B) / (C − D)

④ ~~A B D + * E | F G H K | + * -~~

~~A (B + D) * E | F G H K | + * -~~

~~A * (B + D) E | F G H K | + * -~~

(10)

Convert the foll. to postfix.

① $(A+B) * (C-D)$

$(AB+) * (CD-)$

$AB+ CD- *$

⑤ $A \$ B * C - D + E | F | (G+H)$

$(AB\$) * C - D + E | F | (GH+)$

$((AB\$C) * D -) + (EF|) | (GH+)$

$(AB\$C * D -) + (EF| GH+|)$

$AB\$C * D - EF| GH+| +$

③ $((A+B) * C - (D-E)) \$ (F+G)$

$((AB+) * C - (DE-)) \$ (FG+)$

$((AB+C*) - (DE-)) \$ (FG+)$

$(AB+C*DE--) \$ (FG+)$

$AB+C*DE-- FG+\$$

④ $A - B | (C * D \$ E)$

$A - B | (C * DE\$)$

$A - B | (CDE\$*)$

$A - BCDE\$*|$

$ABCDE\$*|-$

Algorithm to Convert infix expr. to postfix expr

Let E be an infix expr. with '\0' be the end of string. This algorithm uses a stack[N] and variable top.

Symbol be a general name given for various components of the expression. i.e '(', ')', operators and operands.

1. Initially stack is empty.
2. Push symbol '#' onto stack, whose precedence is 0.
3. Pick the Symbol from the infix expr.
4. If the symbol is an operand, then put the symbol onto postfix string.
5. If the symbol is an operator, then

Check while precedence(symbol) ≤ prec(stack[top])
    pop symbol from stack and push into postfix string
Then else
    push the symbol onto the stack.

6. If the symbol = '('
    push the symbol onto stack.

7. If sy
whi
{

8. If
wh
{

Tracing

Index	
0	
1	
2	
3	
4	
5	
6	

postfix expr

'\0' be the

a stack [N]

ven for
on. ie

bihose

infix expr.

d, then

string.

or, then

ack [top])

od

stack.

tack.

7. If symbol = ')' Then

while Stack[top] <> '('

{

pop symbol from Stack

push onto postfix expr

} → Pop the symbol

8. If symbol = '\0' Then

while (Stack is not empty)

{

pop symbol from Stack

store it into postfix expr

}

Tracing of algo. on a+b*c)

Index	Symbol	postfix	Stack	comments
0	#			push '#' onto stack
1	a	a		push onto postfix (operand)
2	+			push '+' onto stack
3	b	a b		push o onto postfix
4	*			Push * onto Stack
5	c	a b c		push c onto postfix
6	'\0'	a b c * +		pop from Stack & place it onto postfix

Precedence Table

Operators	precedence	
# (	0	lowest
+ −	1	
* /	2	
^ $	3	Highest
)		

Trace of the algo. on (a+b)*c

0	1	2	3	4	5	6	7
(	a	+	b	)	*	c	'0'

Index	Symbol	postfix	stack	Comment
0	(		#(	push # onto stack & push it onto stack
1	a	a		place it on postfix
2	+	a	#(+	push onto stack
3	b	ab	#(+	place a onto postfix
4	)	ab+	#	ppp from stack & place it on postfix
5	*	ab+	#*	push * onto stack
6	c	ab+c	#*	place c onto postfix
7	'0'	ab+c*		

Program

```
void ma
{
char in
int i,
clrscr(
Printf(
gets(
push
for (i=
{
if (
po
Else
}
```

11	$	AB+CD-	# * $	push $ onto st
12	E	AB+CD-E	# * $	Place E on post
13	*	AB+CD-E$*	# *	
14	F	AB+CD-E$*F		
15	'\0'	AB+CD-E$*F*		

(13)

(2) ((A+B) * C − (D−E)) $ (F+G)

Index	Symbol	postfix	stack
0	(		#(
1	(		#((
2	A	A	#((+
3	+		
4	B	AB	#(
5	)	AB+	#(*
6	*		
7	C	AB+C	#(−
8	−	AB+C*	#(−(
9	(	AB+C*D	
10	D	AB+C*D	#(−(−
11	−	AB+C*DE	#(−(−
12	E	AB+C*DE−	#(−
13	)	AB+C*DE−−	#
14	)		#$
15	$	AB+C*DE−−F	#$(
16	(		#$(+
17	F	AB+C*DE−−FG	
18	+		#$
19	G	AB+C*DE−−FG+	
20	)		
21	'\0'	AB+C*DE−−FG+$	

ment
onto stack
onto stack
in postfix
onto stack
in postfix
symbols from
till u get '('
'('
onto stack
onto stack
in postfix
onto stack
in postfix
till u get 'E'
also

Queue :->

The term queue is familiar to us in day to day life

-> people standing in a queue to board the bus.
-> people standing in a cinema hall to get the ticket

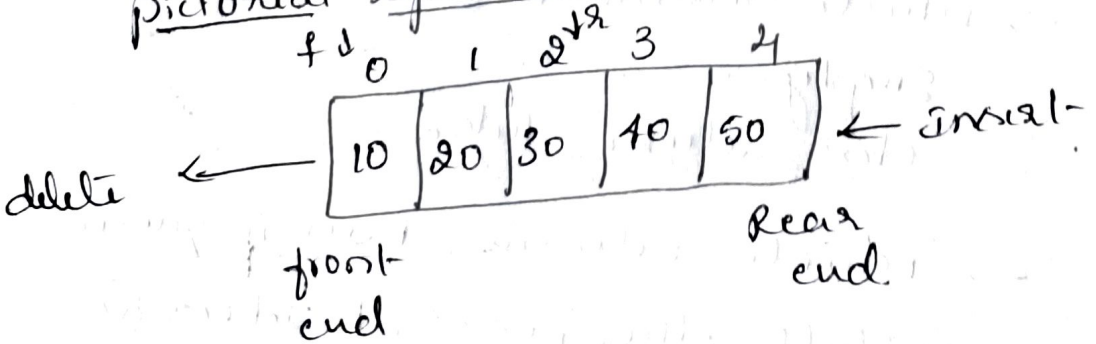
In any situation a person just arrives will stand at the end of the queue and the person who is at the front of the queue is the first person to board the bus @ to get the ticket. The same concept used in the field of computer science also.

Definition:- A queue is an ordered collection of items from which items may be deleted at one end called the front of the queue and items may be inserted at the other end called rear of the queue

* Since the first item inserted is the first item to be deleted from queue. So queue is called First in First out (FIFO)

* Elements always inserted at Rear
* Elements always deleted from the front-end

Pictorial representation:-



* Here the front end is denoted by f and rear end is denoted by r.
* The first item inserted is 10, second item inserted is 20 and the third item inserted is 30 ....
* Any new element to be inserted into this queue has to be inserted towards right
* The first item to be deleted from the front of the queue i.e 10.

So it is clear from the operation performed on queue that first item inserted is the first item to be deleted out from the queue. So queue is called First in first out data structure

operations performed on queue;s

There are three primitive operations that can be performed on queue

→ Insert an item into queue
→ Delete an item from queue
→ Display the contents of queue.

other useful operations

*qempty() → which returns true if queue is Empty
else return false

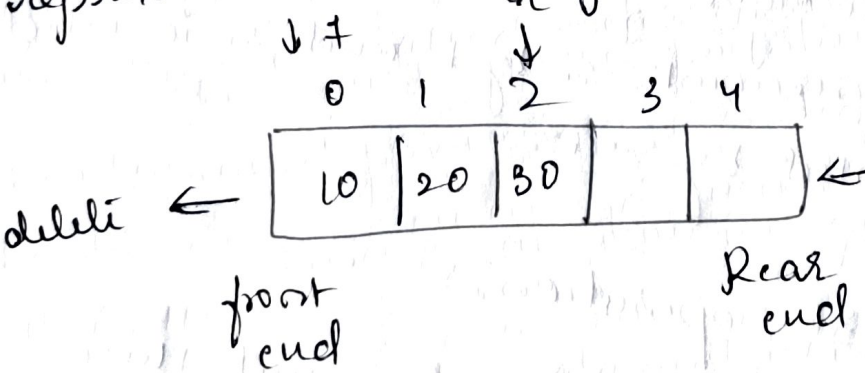
*qfull() → which returns true if queue is full, otherwise it returns false

Different types of queues

* Queue (ordinary queue)
* circular queue
* Double ended queue
* priority queue

Queue (ordinary queue) is

This queue operates on first come first serve basis. Items will be inserted from one end they are deleted at the other end in the same order in while they are inserted. Here the first element inserted is the first element to be deleted. A queue can be represented as in figure



The operations are

* Insert an item at the rear end
* Delete an item at the front end
* Display the contents of queue

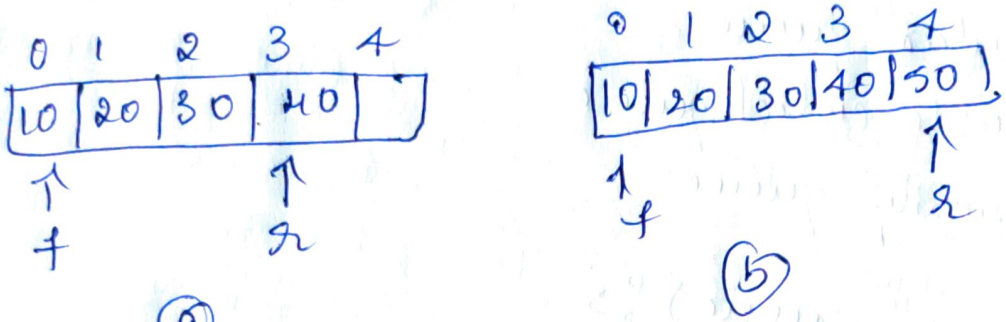
Representing Queue in 'C' is

Queue can be represented

* using arrays
* using structures

Implementation of queue's

Consider a queue with QUEUE-SIZE as 5 and assume 4 items are present as in figure



(a) (b)

Fig:- To insert an item 50.

from the figure it is clear that at most 5 elements can be inserted into the queue. Any new item to be inserted is from the rear end of the queue. So, if an item 50 has to be inserted, it has to be inserted to the right of item 40, i.e at q[4]. It is possible if we increment r by 1 and is inserted into the position pointed by rear.

Algorithm for insert operation:

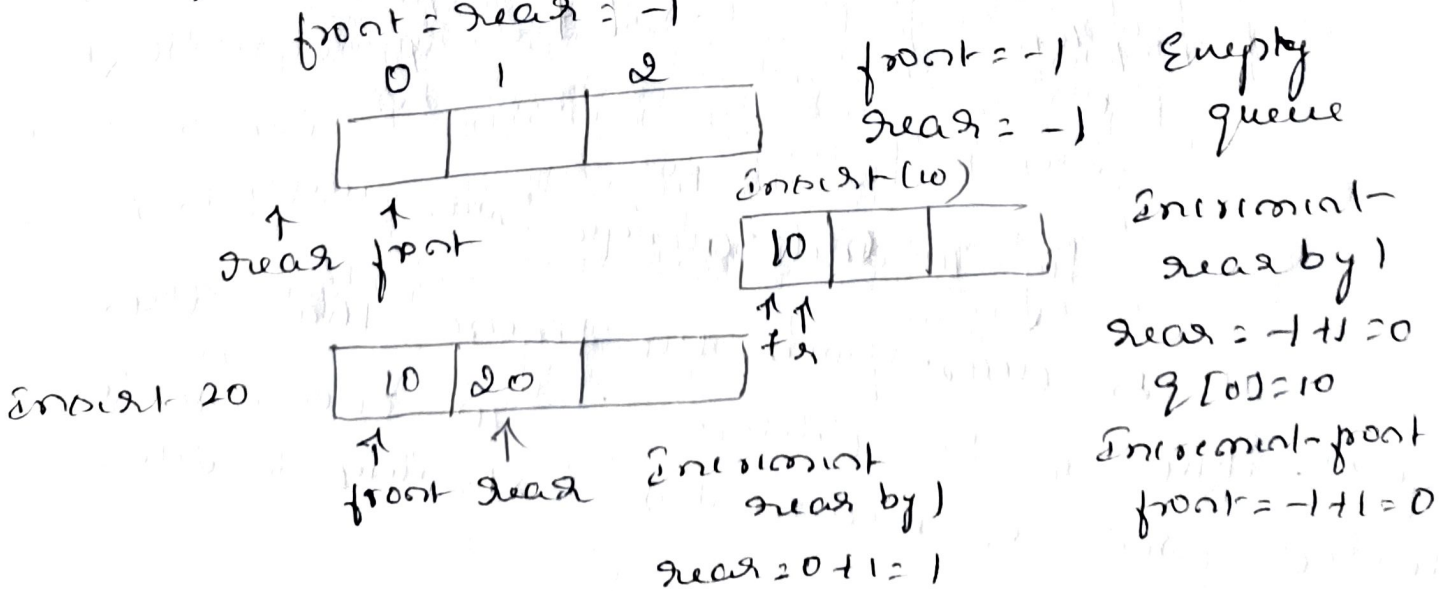
1) check if rear = Size - 1, if it is equal then queue is full ⇒ overflow. otherwise increment rear by 1 and insert the element at that position.

2) check if front = -1 then assign front = 0. It indicates the first element of the queue is inserted

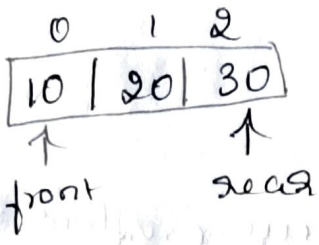
Implementation

```
void qinsert()
{
int x;
if (rear == SIZE-1) /* queue overflow */
{
printf("Queue is overflow \n");
return;
}
printf("Enter the element to be inserted \n");
scanf("%d", &x);
rear = rear+1;
q[rear] = x;
if (front == -1)
front++;
}
```

Tracing:- consider a queue SIZE = 3



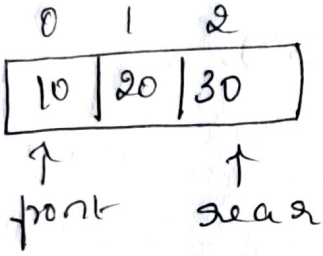
insert (30)



Increment rear

rear = 1 + 1 = 2

Insert (40)



if rear == SIZE - 1

2 = 3 - 1

2 == 2

queue overflow

Delete from front end;

The first item to be deleted from the queue is item, which is at the front end of the queue.

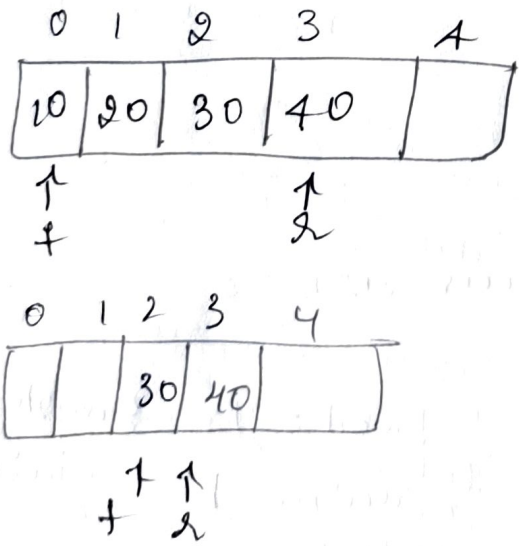


fig:- To delete an item from queue

It is clear from the fegure that the first item to be deleted is 10 once item is deleted front is incremented next item to delete is 20

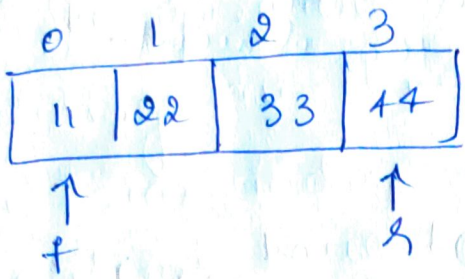
Algorithm:-

1) check if (front = -1) true, queue is empty
2) underflow false, delete the element- & increment the front-
3) check if (front > rear)
   True, assign front = rear = -1
   This indicate that the last element to be deleted

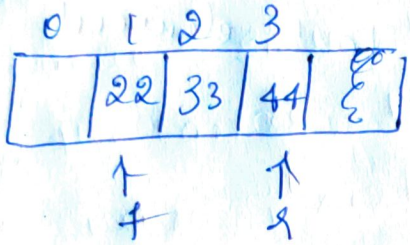
Implementation

```
void delete()
{
if (front == -1) /* check queue is empty or no */
{
printf("Queue underflow \n");
return;
}
printf("deleted element- is %d", q[front]);
front++; /* Increment the front- */
/* If the last- element- is deleted */
        if (front- > rear)
            front = rear = -1;
}
```

Trace of the function Qdelete()

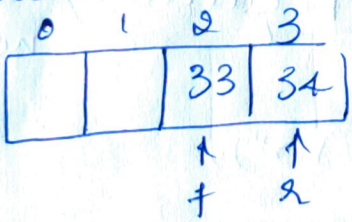


delete



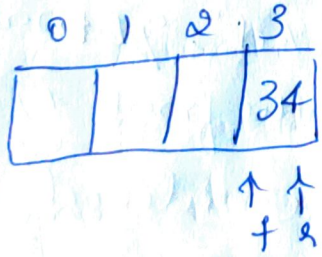
f = 1

delete



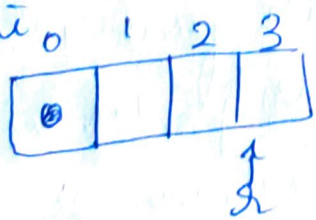
f = 2

delete



f = 3

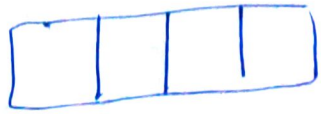
delete



f = 4

f > r i.e 4 > 3

f = r = -1



Display operation:-

It is used to display all the elements in a queue from front to rear.

Algorithm:-

check if the queue is empty

i.e if (front == -1) then Q is empty, otherwise display all elements from front to rear.

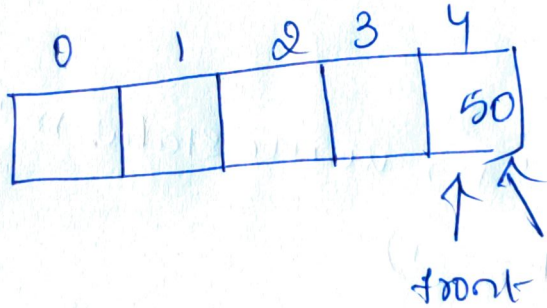
Implementation

```
Display()
{
int i;
printf("\n Queue status \n");
if (front == -1)
{
printf("Q is empty \n");
return;
}
for(i=front; i<=rear; i++)
printf("%d \n", q[i]);
}
```

Disadvantages of queues

The operations such as inserting an element and deleting an element from queue works perfectly until the rear index reaches the end of the array. If some items have been deleted from the front end, there will be some empty space in the beginning of the queue. Since the rear index points to the end of the array, the queue is thought to be full and no more insertions are possible

Example:- consider the queue shown in fig:-



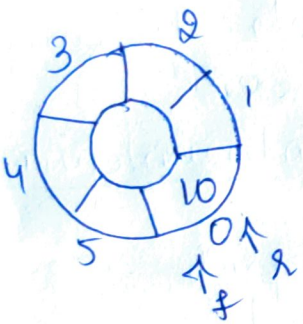
This situation will arise when 5 elements say 10, 20, 30, 40 and 50 are inserted and then deleting first four items 10, 20, 30 and 40. If we say to insert an element say 60, since r has the value QUEUE-SIZE-1 we get an overflow condition and it is not possible to insert any element. Even though the queue is not full it is not possible to insert any item

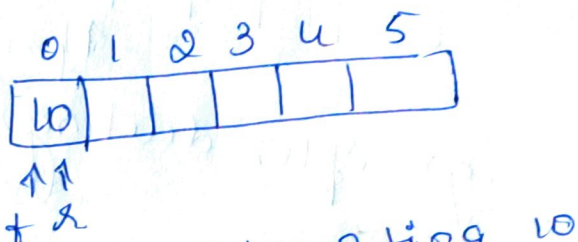
To overcome this disadvantage circular queue is used

Circular Queue:

To overcome the disadvantage of ordinary queue, circular queue is used. In a circular queue the elements of a queue can be stored efficiently in an array so as to "wrap around". So that end of the queue is followed by front of queue. The pictorial representations of circular queue is given by as in figure

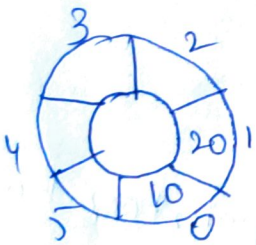
1)

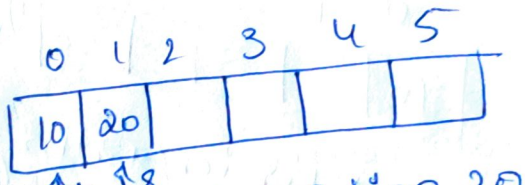




(a) After inserting 10

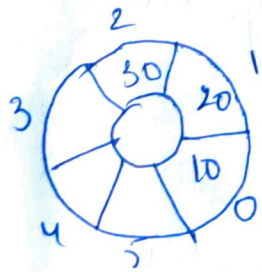
2)

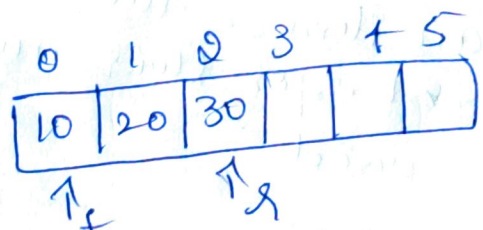




(b) After inserting 20

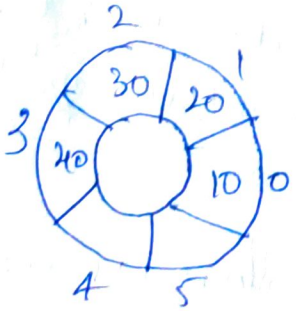
3)

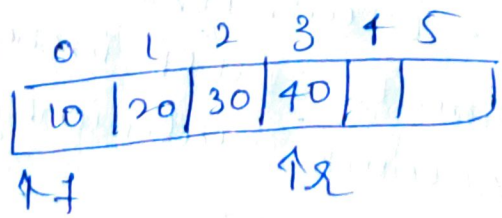




(c) After inserting 30

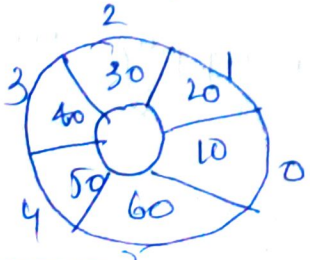
4)

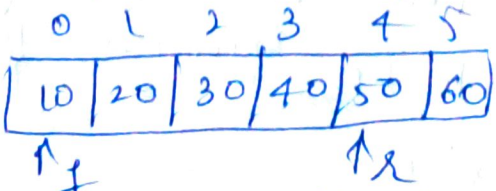




(d) After inserting 40.

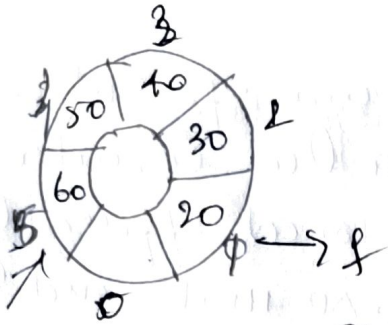
5)

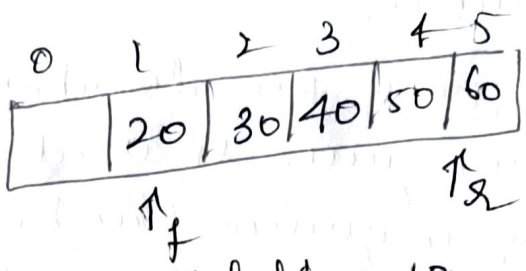




(e) After inserting 50 and 60

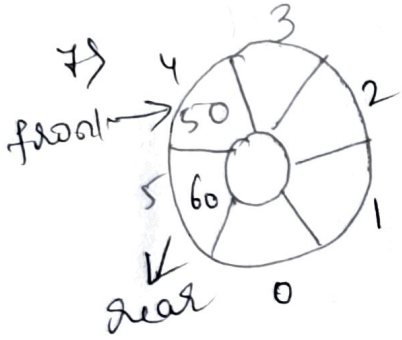
6)

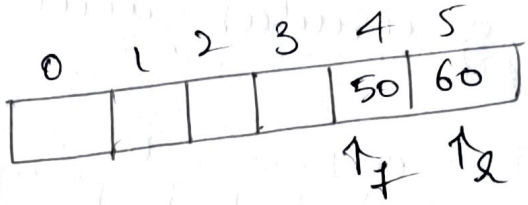




(f) After deleting 10

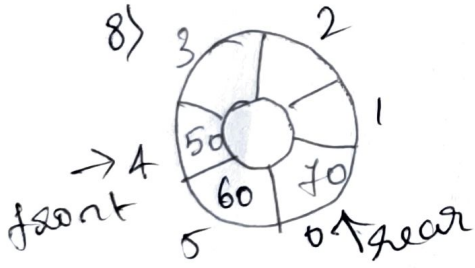
7)

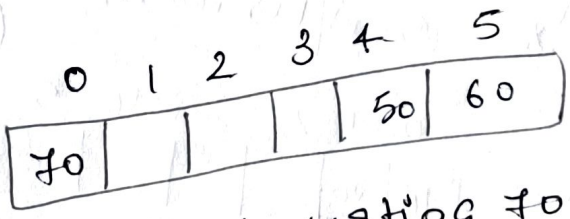




(g) After deleting 20, 30, 40.

8)





(h) After inserting 70

To insert an item, the rear pointer has to be incremented first. for this any of the two statements can be used.

rear = rear + 1

(or)

rear = (rear + 1) % QUEUE-SIZE

Both statements will increment rear by 1 But we prefer the second statements instead of the Statement rear = rear + 1.

Consider the situation g. Now we are trying to insert 70 rear = 5. if we apply rear = rear + 1 rear = 6

Because this is a circular queue it should point to 0. This can achieved using the

rear = (rear + 1) % QUEUE-SIZE

After executing this

rear = (5 + 1) % 6

rear = 6 % 6

rear = 0

In this approach to check for overflow & underflow, we use a variable count that contains the number of items in the queue

If an item is inserted increment count by 1

If an item is deleted decrement the count by 1

operations on circular queue is

* insert
* delete
* display

insertion operation

```
void cqinsert()
{
  int x;
  if (count == MAXSIZE)
  {
    printf("queue is full\n");
    return;
  }
```

```
printf("Enter value in");
scanf("%d", &x);
rear = (rear+1) % MAXSIZE;
                /* increment rear by 1 */

cq[rear] = x;
count++;        /* As you insert increment the
                   count by 1 */
if (front == -1)
  front++;
}
```

Algorithm:- 1. check if the queue is full
i.e if (count == MAXSIZE)
True, print the appropriate message
else increment the rear pointer
rear = (rear+1) % MAXSIZE
insert the element

2. update count by 1. The variable count contains the number of elements in the queue.

Delete operation:-

```
void cdelete()
{
  if (count == 0)   /* check for queue empty */
  {
  printf("queue empty underflow in");
    return;
    }
  front = (front+1) % MAXSIZE;
```

printf ("\n element deleted is %d", cq[front]);
count--;
}

Algorithm:-

1. check if the queue is empty
   i.e if (count == 0)
   True print the appropriate message
   False Access an item which is at the front end by specifying cq[front] and increment the front by using
   front = (front + 1) % MAXSIZE
2. As an item is deleted from the queue decrement count by 1.

Display operation :->

```
void display()
{
  int i, j;
  printf("\n Q status");
  if (count == 0)
  {
    printf("\n queue is empty \n");
    return;
  }
  i = front;
  for (j = 1; j <= count; j++)      /* count contains the no of items in the queue */
```

```
    i = (i+1) % MAXSIZE;
    printf("%d\n", cq[i]);
  }
}
```

Algorithm:-

1 check if queue is empty

if(count == 0)

True print the appropriate message

false if queue is not empty, elements in the queue should be displayed from the front end identified by front to the rear end identified by rear this can be achieved by initializing the index variable i, to the front end and increment i each time using the

i = (i+1) % MAXSIZE

This can be done count times.

Double ended Queue :- (DE Queue)

A double ended queue is a linear list in which elements can be added & removed at either end but not in the middle.

There are two variations of dequeue

* an input-restricted dequeue
* an output-restricted dequeue

These are intermediate b/w a dequeue and a queue

Input restricted queue is a dequeue which allows insertion at only one end of the list - but allows deletion at both end of the list.

Output restricted queue - is a dequeue which allows deletion at only one end of the list - but allows insertions at both ends of the list -

Dequeue -

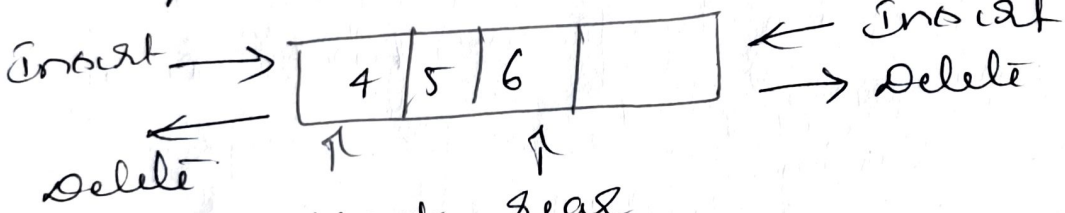


fig - pictorial representation of dequeue Inserting both end and deleting both end

Input restricted queue -

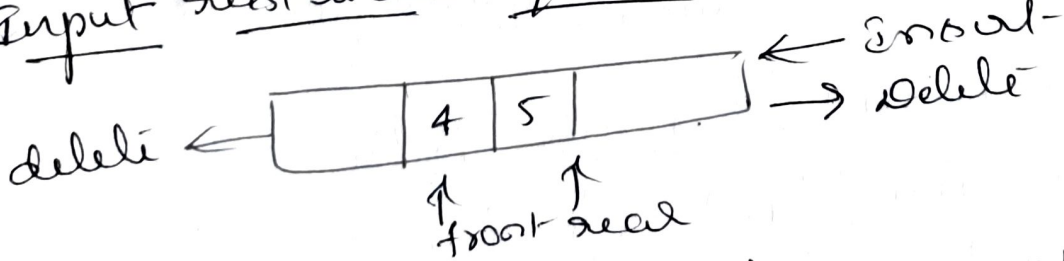


fig - pictorial representation of dequeue Inserting at only one end (rear)
Deletion at both ends del-front()
del-rear()

Output restricted queue:-

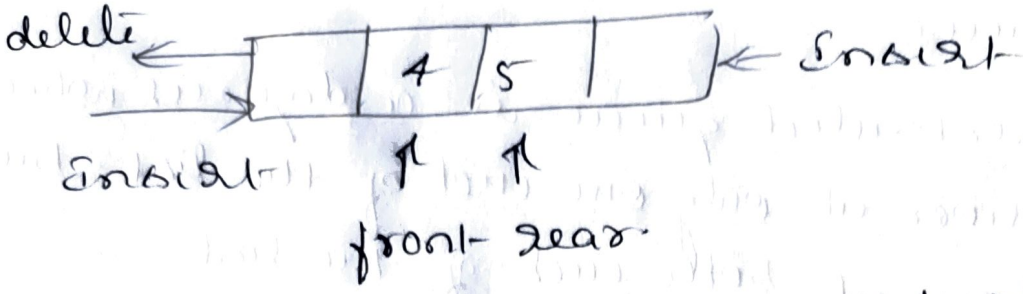


fig:- pictorial representation Dequeue

Inserting at both end [Ins-rear, Ins-front)

Deleting at only end (front) → del-front)