Data Structures

Module - 2

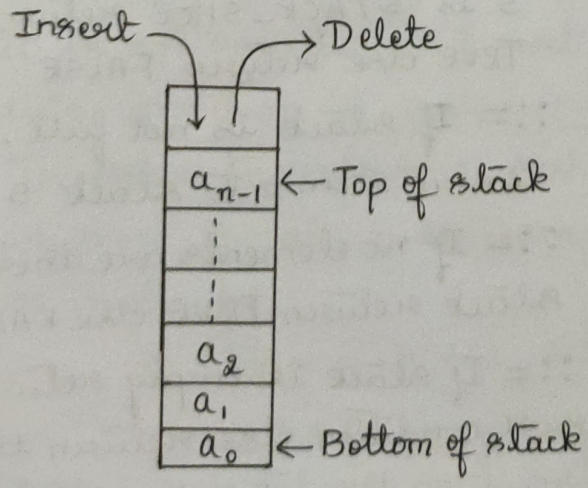
Syllabus :-

Stacks :- Definition, Stack Operations, Array Representation of stacks, Stacks using Dynamic Arrays, Different representation of expression, Stack Applications : Infix to Postfix conversion, Infix to Prefix conversion, evaluation of postfix expression, Recursion.

Queues :- Definition, Array Representation of Queues, Queue Operations, Circular Queues, Queues and Circular Queues using Dynamic Arrays, Dequeues, Priority Queues.

Definition :-

- A stack is an ordered list in which insertions (pushes) and deletions (pops) are made at one end called the top.
- Elements are inserted from one end and elements are deleted from the same end.
- Using this approach, the Last element Inserted is the First element to be deleted Out, and hence, stack is also called Last in First Out (LIFO) data structure.
- Given a stack $S = (a_0, \ldots, a_{n-1})$, where a_0 is the bottom element, a_{n-1} is the top element, and a_i is on top of the element a_{i-1}, $0 < i < n$.



- Elements are inserted into the stack in the order $a_0, a_1, a_2, a_3 \ldots a_{n-1}$
- That is, we insert a_0 first, a_1 next and so on.
- The item a_{n-1} is inserted at the end. Since, it is on top of the stack, it is the first item to be deleted.

The various operations performed on stack are:-

1) **Insert** :- An element is inserted from top end. Insertion operation is called PUSH operation.
2) **Delete** :- An element is deleted from top end only. Deletion operation is called POP operation.
3) **Overflow** :- checks whether the stack is full or not.
4) **Underflow** :- checks whether the stack is empty or not.

Stack Operations :-

Abstract Datatype stack or ADT Stack :-

- An ADT stack is the one that shows the various operations that can be performed on stack.
- An ADT can be written as:

ADT Stack is:

Object : A set of zero or more elements

Functions : Parameters used : STACK_SIZE ∈ positive integer
S ∈ stack, item ∈ element.

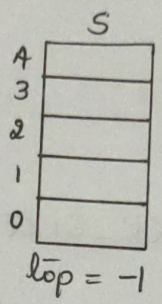
Return_type	Function_Name	Operations
1) stack	Create(STACK_SIZE)	::= creates an empty stack whose maximum size is STACK_SIZE
2) Boolean	IsFull(S, STACK_SIZE)	::= If no of elements in stack S is STACK_SIZE return TRUE else return FALSE
3) Stack	Push(S, item)	::= If stack is not full, insert item onto stack S
4) Boolean	IsEmpty(S)	::= If no elements are there in stack return TRUE else FALSE
5) Element	Pop(S)	::= If stack is empty return error condition else return the element on the top of the stack 'S'

Implementation of stacks using the function defined in ADT Stack :-

1) Create() :-

- Create stack function can be implemented as a single dimensional array i.e,

```
#define STACK_SIZE 5
int S[STACK_SIZE];
int top = -1;
```



- Here, STACK_SIZE is a symbolic constant and specifies the maximum number of elements that can be inserted into the stack.
- S[STACK_SIZE] is an array to hold the elements of the stack
- Variable 'top' holds the index of the topmost element.

If item is the element that is inserted into the stack for the 1st time, then we write

```
S[0] = item
  ↳ top
```

- Value of top is 0 when an item is inserted for the first time. So, before inserting the item, the value of top must be -1.

2) IsFull() :-

- checks the overflow condition for the stack.
- Inserting an item is possible only if the stack is not full, it is not possible to insert any element into the stack and this condition is called stack overflow.
- In order to perform a push operation we need to check whether the stack is completely filled with elements or not. i.e,

```
if (top == STACK_SIZE-1)
{ printf("stack full - overflow");
  return 1; else return 0; }
```

3) Push( ) :-

- Inserts an element onto the stack.
- We first checks whether sufficient space is available in the stack. If available then increments the value of top by 1, and then inserts the item into the stack.

```
Void Push()
{
  int item;
  if (top == STACK_SIZE-1)
  {
   printf("stack overflow");
   return;
  }
   printf("Enter an item\n");
   scanf("%d", &item);
   s[++top] = item;
}
```

`++top` = `top = top + 1;`
`s[++top] = item;` = `s[top] = item;`

4) IsEmpty ( ) :-

- Before we delete an item from the stack, we must first check whether there are any element in the stack i.e,

```
if (top == -1)
{
  Printf("stack is empty");
  return;
}
```

- top = -1 indicates that the stack is empty.

5) Pop( ) :-

- To delete an item from the stack.
- Only one item can be deleted at a time and item has to be deleted only from top of the stack.

```
Void pop()
{
  if (top == -1)
  {
    printf("stack underflow\n");
    return;
  }
  printf("Deleted item = %d", s[top--]);
  return s[top--];
}
```

6) Display():-

- Prints the content of the stack.
- If the stack already has some elements, all those elements are displayed one after the other from bottom to top.

```
Void display()
{
  int i;
  if (top == -1)
  {
    Printf("stack is empty");
    return;
  }
  printf("contents of the stack are:");
  for (i=0; i<=top; i++)   → bottom to top display
  for (i=top; i>=0; i--)   → top to bottom display
    printf("%d\n", s[i]);
}
```

Write a program to implement stacks using arrays:-

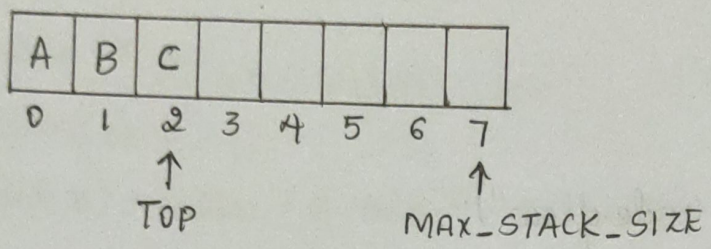
```
#include <stdio.h>
#define SIZE 5
int s[SIZE], top = -1;
Void Push();
Void Pop();
Void display();
int main()
{
 int ch;
 for (;;)
{
  Printf("1. Push, 2. Pop, 3. Display, 4. Exit\n");
  scanf("
  printf("Enter your choice");
  scanf("%d", &ch);
  switch (ch)
  {
   case 1: push(); break;
   case 2: pop(); break;
   case 3: display(); break;
   Default: printf("Invalid choice");
            exit(0);
  }
 }
  return 0;
}

Void push()
{
 int item;
 if (top == SIZE-1)
{
 printf("Stack overflow");
 return;
}
```

```
    printf("Enter an item to be inserted\n");
    scanf("%d", &item);
    s[++top] = item;
}

Void pop()
{
  if (top == -1)
{
  printf("stack underflow");
  return;
}
  printf("Deleted item = %d", s[top--]);
  return s[top--];
}

Void display()
{
  int i;
  if (top == -1)
{
  printf("stack is empty");
  return;
}
  printf("contents of the stack are");
  for (i=0; i<=top; i++)
  printf("%d\n", s[i]);
}
```

Array Representation of Stacks :-

- Stacks may be represented in the computer in various ways such as one-way 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 gives maximum number of elements that can be stored in stack.
- Stack can be represented using linear array as shown below:



Stacks Using Dynamic Arrays :-

- Array is used to implement stack but the bound MAX_STACK_SIZE should be known during compile time.
- Size of bound is impossible to alter during compilation. Hence this can be overcome by using dynamically allocated arrays for the element and then increasing the size of the array as needed.

```
#include <stdio.h>
#include <stdlib.h>
int stacksize = 1;
int *s, top = -1;
s = (int *) malloc (stacksize * sizeof (int));
Void push()
{
  if (top == stacksize -1)
  {
    printf ("stackfull");
    stacksize++;
  s = (int *) realloc (s, stacksize * sizeof (int));
  }
    s[++top] = item;
}
```

```
Void pop()
{
 if (top == -1)
 {
  printf("Stack underflow");
  return;
 }
 printf("Item deleted = %d\n", s[top--]);
 stacksize--;
 s = (int *) realloc (s, stacksize * sizeof(int));
}
Void display()
{
 int i;
 if (top == -1)
 {
  printf("Stack is empty");
  return;
 }
 printf("Contents of the stack are:");
 for (i=0; i<=top; i++)
  printf("%d\n", s[i]);
}
int main()
{
 int ch;
 for(;;)
 {
  printf("1. Push, 2. Pop, 3. Display, 4. Exit\n");
  printf("Enter your choice:");
  scanf("%d", &ch);
  switch (ch)
  {
   case 1: push(); break;
   case 2: pop(); break;
   case 3: display(); break;
```

```
    default : printf("Invalid choice");
              exit(0);
  }
  }
   return 0;
}
```

Different Representation of Expressions :-

Expression :-

- It is a sequence of operators and operands that reduces to a single value after evaluation.

Eg :- $X = a/b - c + d * e$

operators are → +, −, *, /

Operands are → a, b, c, d, e

An arithmetic expression can be represented in 3 types :

1) Infix expression
2) Prefix expression or Polish notation
3) Postfix expression or Reverse Polish notation

1) Infix expression :-

- In this expression, the binary operator is placed in between the operands. The expression can be parenthesized or un-parenthesized.

Eg :- a + b

operand1 ← + → operand2

operator

2) Prefix expression :-

- In this expression, the operator appears before its operand

Eg :- +ab

3) Postfix expression :-

- In this expression, the operator appears after its operands

Eg :- ab +

Precedence of the operators :-

- First problem with understanding the meaning of expressions and statements is finding out the order in which the operations are performed.
- Each operator is associated with priority value.
- Based on the priority, expressions are evaluated.
- Priority of each operator is pre-defined in C language.
- Pre-defined priority given to each operator is called precedence of operator.

Associativity of operators :-

- During evaluation, if 2 or more operators have same precedence then precedence rules are not applicable.
- We go for associativity of operators.
- Order in which the operators with same precedence are evaluated in an expression is called associativity of the operator.

- Operators are arranged from highest to lowest precedence in the given table.
- Operators with highest precedence are evaluated first.
- Last column in the table indicates how the operators having the same precedence should be evaluated.

L→R indicates evaluation has to be done from left to right.

R→L indicates evaluation has to be done from right to left.

Precedence of Various operators:-

Operator	Description	Precedence	Associativity
() []	Parenthesis Array element	15	L→R
++, -- ! ~ +, - & sizeof	Increment, Decrement Logical Not One's complement Unary plus, Unary minus Address operator size (in bytes)	14	R→L
*, /, %	Multiplicative operators	13	L→R
+, -	Binary Add or Subtract	12	L→R
<<, >>	Left shift, Right shift	11	L→R
<, <=, >, >=	Relational operator	10	L→R
==, !=	Equality operator	9	L→R
&	Bitwise AND	8	L→R
^	Bitwise XOR	7	L→R
|	Bitwise OR	6	L→R
&&	Logical AND	5	L→R
||	Logical OR	4	L→R
? :	conditional operator	3	R→L
=, +=, -=, |=, *=, %=	Assignment operator	2	R→L
,	Comma operator	1	L→R

Applications of Stacks :-

1) Conversion of expressions :-

- We write mathematical expression in a program using infix expression.
- Compiler converts the infix expressions into postfix expressions.

2) Evaluation of expressions :-

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

3) Recursion :-

- A function which calls itself is called recursive function. Stacks can be used in recursion.

- Computer usually evaluates an arithmetic expression in 2 steps :-

1) First, it converts the expression (infix) into a postfix notation.
2) Second, it evaluates the postfix expression.

Infix Expression into Postfix Expression :-

Algorithm :-

Steps :-

1) Push # into the stack.
2) Scan the expression from left to right, repeat step ③ to step ⑥ for each element.
3) If an '(' parenthesis is encountered, push it onto stack
4) If an operand is encountered, add it onto postfix P.
5) If an operator is encountered then
   a) Check whether the precedence of the current operator on

top of the stack is greater than or equal to the precedence of the scanned operator. If so, then go on popping all the operators from the stack and place them in postfix P.

b) Otherwise push the scanned operator onto stack.

6) If an ')' parenthesis is encountered, then go on poping all the items from the stack and place them in postfix expression till we get the matching left '(' parenthesis

Note:- Do not push the '(' left parenthesis onto postfix P.

7) Check if any operator or symbols are present in the stack If so, then pop them and place it into postfix P.

C function to convert from infix expression into postfix expression :-

```
Void infix_postfix ( char infix [], char postfix [])
{
  int top;
  int j, i;
  char s[30], symbol;
  top = -1;
  s[++top] = '#';
  j = 0;
 for (i=0; i < strlen(infix); i++)
  {
    symbol = infix [i];
   while (F(s[top]) > G(symbol))
   {
     postfix[j] = s[top--];
     j++;
   }
```

```
    if (F(s[lop]) != G(symbol))
    s[++lop] = symbol;
    else
     lop--;
  }
  while (s[lop] != '#')
  {
    postfix [j++] = s[lop--];
  }
    postfix [j] = '\0';
}
```

Convert the following infix expressions into postfix expression using stack :-

1) (A + (B - C) * D)

Stack	s[top]	Symbol	Operation	Postfix
#	#	(	push (	
#(	(	A	pop A	A
#(	(	+	push +	A
#(+	+	(	push (	A
#(+(	(	B	pop B	AB
#(+(	(	-	push -	AB
#(+(-	-	C	pop C	ABC
#(+(-	-	)	pop -	ABC-
#(+(	(	)	pop (	ABC-
#(+	+	*	push *	ABC-
#(+*	*	D	pop D	ABC-D
#(+*	*	)	pop *	ABC-D*
#(+	+	)	pop +	ABC-D*+
#(	(	)	pop (	ABC-D*+

∴ Postfix expression for the given infix expression

ABC-D*+

2) (A+B/C*(D+C)-F)

Stack	S[top]	Symbol	Operation	Postfix
#	#	(	push (	
#(	(	A	pop A	A
#(	(	+	push +	A
#(+	+	B	pop B	AB
#(+	+	/	push /	AB
#(+/	/	C	pop C	ABC
#(+/	/	*	pop /, push *	ABC/
#(+*	*	(	push (	ABC/
#(+*(	(	D	pop D	ABC/D
#(+*(	(	+	push +	ABC/D
#(+*(+	+	C	pop C	ABC/DC
#(+*(+	+	)	pop +	ABC/DC+
#(+*(	(	)	pop (	ABC/DC+
#(+*	*	-	pop *	ABC/DC+*
#(+	+	-	pop +, push -	ABC/DC+*+
#(-	-	F	pop F	ABC/DC+*+F
#(-	-	)	pop -	ABC/DC+*+F-
#(	(	)	pop (	ABC/DC+*+F-
#	#			

∴ Postfix expression for the given infix expression

ABC/DC+*+F-

- No 2 operators of same priority can stay together in stack column.
  If they are together, pop the operator which is already stored in stack.

- Lowest priority operator cannot be placed after the highest priority operator in the stack.
  If so, then pop the highest priority operator and then push the lowest priority operator.
- Highest priority operator can be placed after the lowest priority operator in the stack.

3) $(a*b)+c$

Stack	S[top]	Symbol	operation	Postfix
#	#	(	push (	
#(	(	a	pop a	a
#(	(	*	push *	a
#(*	*	b	pop b	ab
#(*	*	)	pop *	ab*
#(	(	)	pop (	ab*
#	#	+	push +	ab*
#+	+	c	pop c	ab*c
#+	+		pop +	ab*c+
#	#			

∴ Postfix expression for the given infix expression ab*c+

4) $(A-B)*(D/E)$

Stack	S[top]	Symbol	operation	Postfix
#	#	(	push (	
#(	(	A	pop A	A
#(	(	-	push -	A
#(-	-	B	pop B	AB
#(-	-	)	pop -	AB-
#(	(	)	pop (	AB-

#	#	*	push *	AB-
#*	*	(	push (	AB-
#*(	(	D	pop D	AB-D
#*(	(	/	push /	AB-D
#*(/	/	E	pop E	AB-DE
#*(/	/	)	pop /	AB-DE/
#*(	(	)	pop (	AB-DE/
#*	*		pop *	AB-DE/*
#	#			

∴ Postfix expression for the given infix expression

AB-DE/*

Infix Expression into Prefix Expression :-

Algorithm :-

Steps :-

1) Reverse the given infix expression.
2) Push # into the stack.
3) Scan the expression from left to right, repeat step ④ to step ⑦ for each element.
4) If an '(' parenthesis is encountered, push it into stack
5) If an operand is encountered, add it into prefix P.
6) If an operator is encountered then
   a) check whether the precedence of the current operator on top of the stack is greater than or equal to the precedence of the scanned operator. If so, then go on popping all the operators from the stack and place them in prefix P.
   b) Otherwise push the scanned operator onto stack.

7) If an ')' parenthesis is encountered, then go on popping all the items from the stack and place them in prefix P expression till we get the matching left '(' parenthesis.

Note :- Do not push the '(' left parenthesis onto prefix

8) Check if any operator or symbols are present in the stack. If so, then pop them and place it into prefix P.

9) Reverse the obtained prefix expression.

C function to convert from infix expression into prefix expression :-

```
Void infix_prefix (char infix[], char prefix[])
{
 int top, i, j;
 char s[30], symbol;
 top = -1;
 s[++top] = '#';
 j=0;
 strrev(infix);
 for (i=0; i<strlen(infix); i++)
 {
  symbol = infix[i];
  while (F(s[top]) > G(symbol))
  {
   prefix[j] = s[top--];
   j++;
  }
  if (F(s[top])!= G(symbol))
    s[++top] = symbol;
  else
    top--;
 }
```

```
while (s[top]! = '#')
{
  prefix [j++] = s[top--];
}
  prefix [j] = '\0';
strrev(prefix);
```

Convert the following infix expressions into prefix expression

1) (A+B)*C-D+F

Reverse of infix expression → F+D-C*(B+A)

Stack	Symbol	Operation	Prefix
#	F	pop F	F
#	+	push +	F
#+	D	pop D	FD
#+	-	pop+, push -	FD+
#-	C	pop C	FD+C
#-	*	push *	FD+C
#-*	(	push (	FD+C
#-*(	B	pop B	FD+CB
#-*(	+	push +	FD+CB
#-*(+	A	pop A	FD+CBA
#-*(+	)	pop +	FD+CBA+
#-*(	)	pop (	FD+CBA+
#-*		pop *	FD+CBA+*
#-		pop -	FD+CBA+*-
#			FD+CBA+*-

Reverse the obtained prefix expression

-*+ABC+FD

2) ((A+(B-C)*D)^E+F)

Reverse of infix expression → (F+E^(D*(C-B)+A))

Stack	Symbol	Operation	Prefix
#	(	push (	
#(	F	pop F	F
#(	+	push +	F
#(+	E	pop E	FE
#(+	^	push ^	FE
#(+^	(	push (	FE
#(+^(	D	pop D	FED
#(+^(	*	push *	FED
#(+^(*	(	push (	FED
#(+^(*(	C	pop C	FEDC
#(+^(*(	-	push -	FEDC
#(+^(*(-	B	pop B	FEDCB
#(+^(*(-	)	pop)-	FEDCB-
#(+^(*(	)	pop (	FEDCB-
#(+^(*	+	pop *, push +	FEDCB-*
#(+^(+	A	pop A	FEDCB-*A
#(+^(+	)	pop +	FEDCB-*A+
#(+^(	)	pop (	FEDCB-*A+
#(+^	)	pop ^	FEDCB-*A+^
#(+	)	pop +	FEDCB-*A+^+
#(	)	pop (	FEDCB-*A+^+
#			FEDCB-*A+^+

Reverse the obtained prefix expression

+^+A*-BCDEF

3) (A+B/C*(D+C)-F)

Reverse of infix expression → (F-(C+D)*C/B+A)

Stack	Symbol	Operation	Prefix
#	(	push (	
#(	F	pop F	F
#(	-	push -	F
#(-	(	push (	F
#(-(	C	pop C	FC
#(-(	+	push +	FC
#(-(+	D	pop D	FCD
#(-(+	)	pop +	FCD+
#(-(	)	pop (	FCD+
#(-	*	push *	FCD+
#(-*	C	pop C	FCD+C
#(-*	/	pop *, push /	FCD+C*
#(-/	B	pop B	FCD+C*B
#(-/	+	pop /	FCD+C*B/
#(-	+	pop -, push +	FCD+C*B/-
#(+	A	pop A	FCD+C*B/-A
#(+	)	pop +	FCD+C*B/-A+
#(	)	pop (	FCD+C*B/-A+
#			FCD+C*B/-A+

Reverse the obtained prefix expression

+A-/B*C+DCF

Evaluation of Postfix Expression :-

Algorithm :-

Steps :-

1) Scan the given expression from left to right.
2) a) If the scanned symbol is an operand, push it onto stack
   b) If the scanned symbol is an operator, pop 2 elements from the stack.
   1st popped element is operand 2 and
   2nd popped element is operand 1
   This can be achieved using statements;
   ```
   op2 = S[top--];
   op1 = S[top--];
   ```
   Then perform the required operation
   ```
   res = op1 op op2;
   ```
   Push the result onto the stack.
3) Repeat step (2) until all the symbols are scanned.
4) Pop out the result and print it.

Evaluate the following given postfix expression :-

1) ABC+*CBA-+*

   Given values are A = 1, B = 2 and C = 3

   ∴ 123+*321-+*

Postfix	Symbol	op2	op1	res = op1 op op2	Stack
123+*321-+*	1				1
23+*321-+*	2				12
3+*321-+*	3				123
+*321-+*	+	3	2	2+3 = 5	15
321-+	*	5	1	1*5 = 5	5
321-+*	3				53
21-+*	2				532

1-+*	1				5321
-+*	-	1	2	2-1=1	531
+*	+	1	3	3+1=4	54
*	*	4	5	5*4=20	20

Result = 20

2) 12+3-21+3$-

Postfix	Symbol	Op2	Op1	res = op1 op op2	Stack
12+3-21+3$-	1				1
2+3-21+3$-	2				12
+3-21+3$-	+	2	1	1+2=3	3
3-21+3$-	3				33
-21+3$-	-	3	3	3-3=0	0
21+3$-	2				02
1+3$-	1				021
+3$-	+	1	2	2+1=3	03
3$-	3				033
$-	$	3	3	3$3=27	0,27
-	-	27	0	0-27=-27	-27

Result = -27

3) 5, 6, 2, +, *, 12, 4, /, -

Postfix	Symbol	Op2	Op1	res = op1 op op2	Stack
562+*124/-	5				5
62+*124/-	6				56
2+*124/-	2				562
+*124/-	+	2	6	6+2=8	58
*124/-	*	8	5	5*8=40	40,
124/-	12				40, 12
4/-	4				40, 12, 4

/ -	/ -	4 3	12 40	12/4 = 3 40-3 = 37	40, 3 37

Result = 37

4) 78+65+*

Postfix	Symbol	op2	op1	res = op1 op op2	Stack
78+65+*	7				7
8+65+*	8				78
+65+*	+	8	7	7+8 = 15	15
65+*	6				15, 6
5+*	5				15, 6 5
+*	+	5	6	6+5 = 11	15, 11
*	*	11	15	15*11 = 165	165

Result = 165

Recursion :-

- It is method of solving the problem where the solution to a problem depends on solution to smaller instances of the same problem.

Recursive function :-

- A function that calls itself during execution.
- This enables the function to repeat itself several times to solve a given problem.

Types of recursion :-

1) Direct recursion
2) Indirect recursion.

Direct Recursion :-

- Recursive function that invokes itself is said to have direct recursion.

Eg :- Factorial function calls itself and hence it is a recursive function.

```
int fact (int n)
{
  if (n==0) return 1;
  else
  return n* fact (n-1);
}
```

direct recursion

Indirect Recursion :-

- Function which contains a call to another function which in turn calls another function, which in turn calls another and so on, eventually calls the first function is called indirect recursion.
- Very difficult to read, understand and find any logical errors in a function that has indirect recursion.

Eq :- Function f1 invokes f2 which inturn invokes f3 which inturn invokes f1 is said to have indirect recursion.

```
Void f1()          Void f2()          Void f3()
{                  {                  {
 ...                ...                ...
 ....               ....               ....
 f2();              f3();              f1();
}                  }                  }
```

Indirect Recursion

- Every recursive call must solve one part of the problem using base case (or) reduce the size of the problem using general case.

Base Case :-

- Special case where solution can be obtained without using recursion.
- Also called as base / terminal condition.
- Each recursive function should have base case.

General Case :-

- In any recursive function, part of the function except base case is called general case.
- This portion of the code contains the logic required to reduce the size of the problem so as to move towards the base case / terminal condition.
- Each time function is called, the size of the problem is reduced.

Recursion can be implemented for the following techniques :-

1) Factorial of a number :-

$$n! = 1.2.3 \text{ ---- } (n-2)(n-1).n$$

Factorial function can be defined as

a) If $n = 0$ $\quad n! = 1$

b) If $n > 1$ $\quad n! = n * (n-1)!$

Eg :- Compute 5!

$5! = 5 * 4!$

$4! = 4 * 3!$

$3! = 3 * 2!$

$2! = 2 * 1!$

$1! = 1 * 0!$

$0! = 1$

} General Case $n! = n * (n-1)!$

} Base Case $n! = 1$ if $n = 0$

Recursive function to find factorial of N:-

```
int fact (int n)
{
  if (n==0) return 1;
  else
  return (n*fact(n-1));
}
```

Program to find factorial of N:-

```
#include <stdio.h>
int fact (int n)
{ if (n==0) return 1;
  else
  return (n*fact(n-1));
}
Void main()
{ int n;
  printf(" Enter n\n");
  scanf(" %d", &n);
  Printf(" Factorial of %d = %d\n", n, fact(n));
}
```

output:-

```
Enter n : 5
Factorial of 5 : 120
```

Algorithm :-

FACTORIAL (FACT, N)

1) If N=0, then set FACT := 1 and return
2) Call FACTORIAL (FACT, N-1)
3) Set FACT := N * FACT
4) Return

2) Fibonacci Series :-

- Fibonacci numbers are a series of no such that each number is the sum of the previous 2 no except the first and second number.
- Fibonacci sequence is as follows:

  0, 1, 1, 2, 3, 5, 8, 13, 21 . . . .

i.e, $F_0 = 0$ and $F_1 = 1$ and each succeeding term is the sum of the 2 preceeding terms.

a) If $n = 0$ or $n = 1$ then $F(n) = n$

b) If $n > 1$ then $F(n) = F(n-1) + F(n-2)$

Recursive definition to find n^{th} fibonacci no can be written as:

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

Eg :- $F(0) = 0$
$F(1) = 1$ } Base case

$F(2) = F(1) + F(0) = 1 + 0 = 1$
$F(3) = F(2) + F(1) = 1 + 1 = 2$
$F(4) = F(3) + F(2) = 2 + 1 = 3$
$F(5) = F(4) + F(3) = 3 + 2 = 5$
} General case

Algorithm :-

FIBONACCI (FIB, N)

1) If N=0 or N=1 then set FIB := N & return
2) call FIBONACCI (FIBA, N-1)
3) Call FIBONACCI (FIBB, N-2)
4) Set FIB = FIBA + FIBB
5) Return

Recursive function to find fibonacci number :-

```
int F(int n)
{
  if (n==0) return 0;
  if (n==1) return 1;
  return F(n-1)+F(n-2);
}
```

C program to display n^{th} fibonacci number :-

```
#include <stdio.h>
int fib(int n)
{ if (n==0) return 0;
  if (n==1) return 1;
  return fib(n-1)+ fib(n-2);
}
void main()
{
 int n;
 Printf("Enter n");
 scanf("%d", &n);
 Printf("fib(%d) = %d\n", n, fib(n));
}
```

output :-

```
Enter n: 6
Fib(6) = 8
```

C program to display n Fibonacci number :-

```
#include <stdio.h>
int fib(int n)
{ if (n==0) return 0;
  if (n==1) return 1;
  return fib(n-1)+ fib(n-2);
}
```

```
Void main()
{
 int i, n;
 Printf("Enter n");
 Scanf("%d", &n);
 Printf("%d fibonacci no are|n", n);
 for(i=0; i<n; i++)
 {
 Printf("fib(%d) = %d|n", i, fib(i));
 } }
```

output :-

Enter n : 6

6 fibonacci no are

fib (0) = 0

fib (1) = 1

fib (2) = 1

fib (3) = 2

fib (4) = 3

fib (5) = 5

Fibonacci series of first 6 numbers.

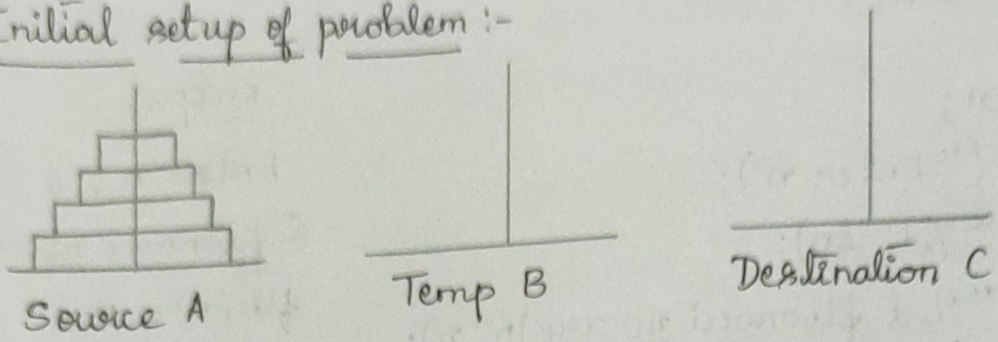
3) Tower of Hanoi :-

- In this problem, there are 3 needles say A, B, C
- Different diameters of 'n' discs are placed one above the other through needle 'A' and the discs are placed such that always a smaller discs are placed above the larger disc.
- 2 Needles B and C are empty.
- All the discs from needle A are to be transferred to needle C using needle B as temporary storage.

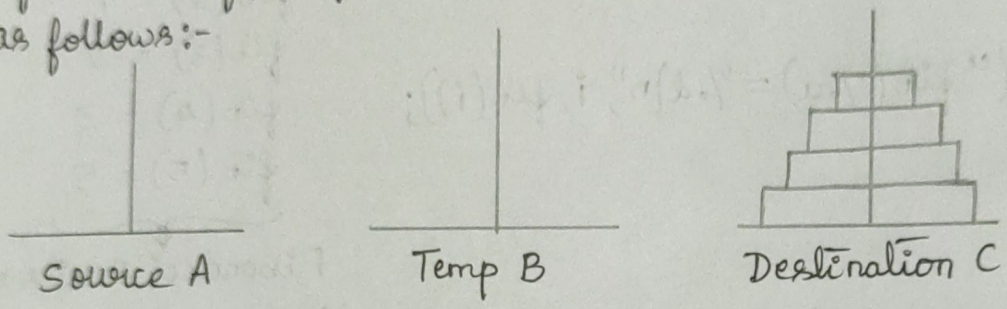
Rules to be followed while transferring the discs are :-

- Only one disc is moved at a time from one needle to another needle.
- Smaller disc is on top of the larger disc at any time.
- Only one needle can be used to for storing intermediate discs.

Initial setup of problem :-



After transferring all discs from A to C, final setup is as follows :-



Base case :-

- Occurs when there are no discs. In such situation we simply return
  i.e, if $n = 0$, return.

General case :-

- Occurs if one or more discs have to be transferred from source to destination.
- If there are 'n' discs, then all 'n' discs can be transferred recursively using 3 steps :

a) Move $n-1$ discs recursively from source to temp
b) Move n^{th} disc from source to destination
c) Move $n-1$ discs recursively from temp to destination

Algorithm :-

TOWER (N, BEG, AUX, END)

1) If N=1 then
   a) Write : BEG → END
   b) Return
   [End of If structure]

2) [Move N-1 discs from BEG to AUX]
   Call TOWER(N-1, BEG, END, AUX)
3) Write : BEG → END
4) [Move N-1 discs from AUX to END]
   Call TOWER (N-1, AUX, BEG, END)
5) Return

Recursive function to implement tower of hanoi :-

```
Void tower (int n, char source, char temp, char destination)
{ if (n==0) return;
 tower (n-1, source, destination, temp);
 Printf (" Move disc %d from %C to %C\n", n, source,
                                        destination);
 tower(n-1, temp, source, destination);
}
```

C program to implement Tower of Hanoi :-

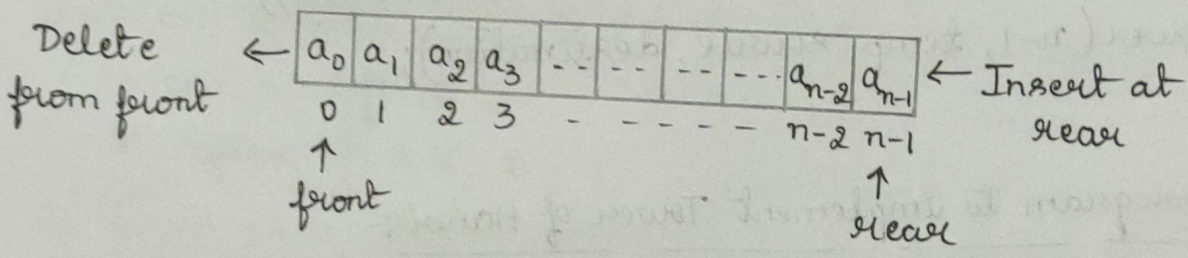
```
#include <stdio.h>
Void tower (int n, char source, char temp, char destination)
{ if (n==0) return;
 tower(n-1, source, destination, temp);
 Printf (" Move disc %d from %C to %C\n", n, source,
                                        destination);
tower(n-1, temp, source, destination);
}
 Void main ()
{
  int n;
 Printf(" Enter the no of discs");
 scanf (" %d", &n);
 tower (n, 'A', 'B', 'C');
}
```

- For $n=3$, there will be 8 moves.
- For $n=4$, there will be 15 moves.

In general, there will be (n^2-1) moves.

Queues :-

- Queue is a linear data structure where elements are inserted from one end and elements are deleted from the other end.
- End at which new elements are added is called rear
- End at which elements are deleted is called front
- Queue works on the principle FIFO (First In First Out) i.e, first element inserted is the first element to be deleted.



Operations performed on queue :-

1) Insert - Element is inserted from rear end.
2) Delete - Element is deleted from front end.
3) Overflow - If queue is full and we try to insert an item, overflow condition occurs.
4) Underflow - If queue is empty and we try to delete an item, underflow condition occurs.
5) Display - Prints/ display the elements present in queue.

Advantages :-

- Used in application and system software development.
- CPU process the jobs in FIFO order.
- Printer prints the file in FIFO.

Types of Queues :-

1) Linear queue (ordinary queue)
2) Circular queue
3) Double Ended queue (Dequeue)
4) Priority queue.

Queue Operations :-

Implementation of linear queues using arrays :

1) Create() :-

- Queue can be created by using a symbolic constant Q_SIZE which specifies the maximum no of elements that can be inserted into the queue.
- This can be defined as

```
# define Q_SIZE 5
```

- q[] is an array which holds the elements of the queue as shown

  q[0] holds the 1st element
  q[1] holds the 2nd element
  q[2] holds the 3rd element & so on.

  It can be defined as

```
int q[Q_SIZE];
```

- Value of front and rear is '0' when an item is inserted for the 1st time from rear end.
- Since, we are inserting from rear end, before inserting the item, value of rear must be '-1' and front is '0'

```
int front = 0;
int rear = -1;
```

- Above statements can also be declared when there are no elements in queue, i.e, when queue is empty.

- So, create() can be implemented as

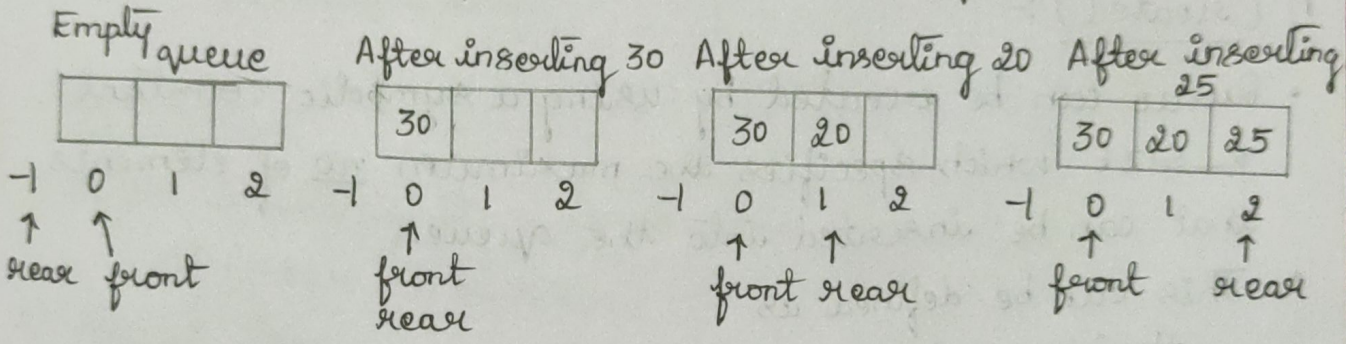
```
#define Q_SIZE 5
int q[Q_SIZE];
int front=0, rear=-1;
```

2) IsFull():-

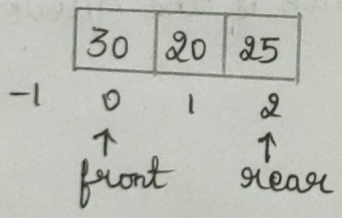
- Elements are inserted from rear end and deleted from front end.

Eg:- Q_SIZE = 3

Items - 30, 20 & 25 are inserted into queue.



- Inserting 50 results in overflow.



- After inserting 3 elements, queue is full and it is not possible to insert any element.
- When queue is full, value of index rear = 2, i.e, Q_SIZE-1
- If rear is same as Q_SIZE-1, then queue is full, we return 1 indicating queue is full else return 0 indicating queue is not full.

Function:-

```
int IsFull()
{
  if(rear == Q_SIZE-1)
  return 1; else return 0; }
```

OR

```
int IsFull()
{ return rear ==
        Q_SIZE-1;
}
```

3) InsertQ() :-

- Before inserting, we check whether sufficient space is available in the queue using IsFull().
- When IsFull() condition fails, we can insert an item into the queue
- To insert an item, we have to increment rear by 1 as shown

```
rear = rear + 1;
```

- Then item can be inserted at rear end of the queue using

```
q[rear] = item;
```

Algorithm :-

1) Before inserting item, first check for overflow
   if (rear == Q_SIZE - 1)
2) Accept item
3) Increment rear by 1. i.e, rear = rear + 1
4) Assign q[rear] = item.
5) End.

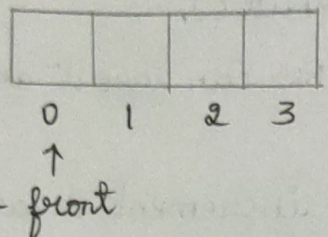
Function to insert an item into queue :-

```
Void InsertQ()
{
 int item, rear = -1;
 if (rear == Q_SIZE - 1)
{
 printf ("Queue Overflow");
 return;
}
 printf ("Enter item \n");
 scanf ("%d", &item);
 rear = rear + 1;
}q[rear] = item;
```

{ ⇒ q[++rear] = item;

4) IsEmpty():-

- Consider a queue where no elements are present.



- Whenever queue is empty, value of rear is '-1' and front is '0' and in this situation value of front is greater than that of rear.
- So, we return 1 if front > rear indicating queue is empty or else return 0.

Function to implement IsEmpty():-

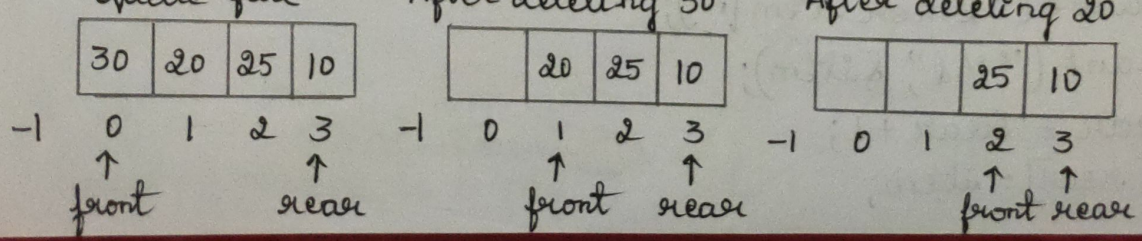
```
int IsEmpty()
{
 if (front > rear)
 return 1;
 else
 return 0;
}
```

OR

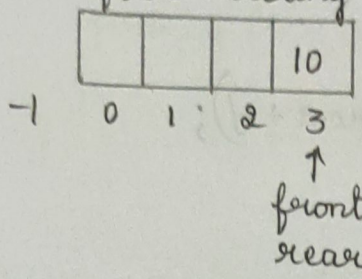
```
int IsEmpty()
{
return front > rear;
}
```

5) DeleteQ():-

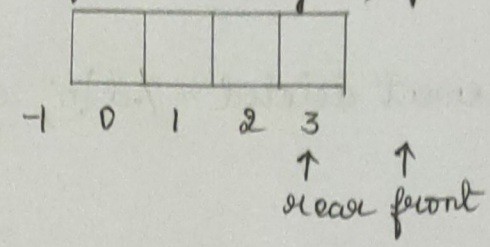
- In a queue, an element is always deleted from the front end.
- Eg:- Assume that 30, 20, 25 & 10 are already inserted into queue. Now all these items can be deleted one after the other from the front end of queue as shown



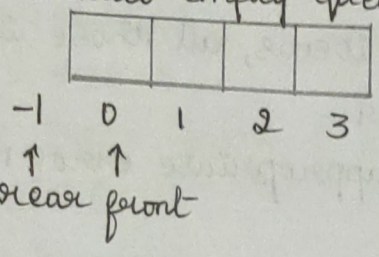
After deleting 25



After deleting 10, queue is empty



Initial empty queue



- As the items are deleted, value of front is incremented by 1 and when queue is empty, the value of front is greater than the value of rear
- Before deleting, we check whether queue is empty using IsEmpty() function.
- When IsEmpty() condition fails, we can delete an item from front end of queue.
- For this, we have to access & return the 1st element & increment value of front by 1.

Algorithm :-

1) check for underflow. i.e, if (front > rear)
2) Delete item i.e, q[front]
3) Increment front by 1. i.e, front = front + 1
4) End.

Function to delete an item from queue :-

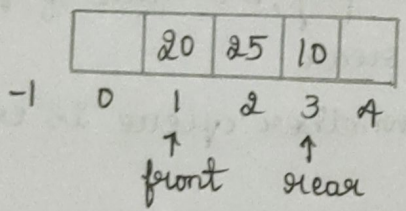
```
Void deletefront()
{ front = 0;
 if (front > rear)
{ printf("Queue is empty");
```

```
return;
}
Printf("Element deleted = %d\n", q[front++]);
}
```

6) Display():-

- If queue already has some items, all those items are displayed one after the other.
- If no items are present, appropriate error message is displayed.

Eg:- Assume that the queue contains 3 elements as shown



- Contents of the queue are displayed from front to rear, so
  First item to be displayed is 20
  Next item to be displayed is 25
  Final item to be displayed is 10
- This can be achieved in general, by using the statement
  printf("%d\n", s[i]);
  where, i = front to rear

Function to display the elements of queue:-

```
Void display()
{ int i;
 if (front > rear)
{ printf("Queue is empty");
 return;
}
 printf("contents of queue\n");
 for(i = front; i<=rear; i++)
```

```
    {
      printf("%d\n", q[i]);
    }
  }
```

Queue implementation using arrays :-

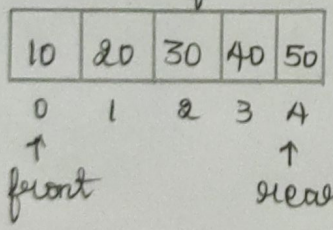
```
#include <stdio.h>
#define Q_SIZE 5
int choice, rear, front, i, item, q[Q_SIZE];
Void InsertQ();
Void deletefront();
Void display();
Void main()
{
 front=0, rear=-1;
 for(;;)
{
 printf("1. Insert\t 2. Delete\t 3. Display\t 4. Exit");
 printf("Enter your choice");
 scanf("%d", &choice);
 switch(choice)
{
 case 1: InsertQ(); break;
 case 2: Delete front(); break;
 case 3: display(); break;
 case 4: exit(0); break;
 default: printf("Invalid choice");
} } }
Void InsertQ()
{
 int rear=-1;
 if(rear==Q_SIZE-1)
{ printf("Queue overflow");
 return;
}
```

```
printf("Enter item");
scanf("%d", &item);
q[++rear] = item;
}
Void deletefront()
{
 int front=0;
if (front > rear)
{
 printf("Queue empty");
 return;
}
printf("Element deleted=%d\n",
        q[front++]);
}
```

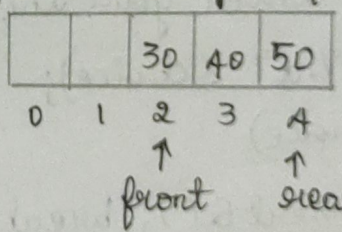
```
Void display()
{
 int i;
 if (front > rear)
{
 printf("Queue Empty");
 return; }
 printf("contents of queue");
for(i=front; i<=rear; i++)
 {
 printf("%d\n", q[i]);
 }
 }
```

Disadvantages of linear queue:-

i) Queue full



ii) After deleting 10 & 20



- Now if we try to insert a new item into queue, we get the message as "Queue Overflow".
- In the above situation, insertion of new item at rear end is denied even if space is available at the front end.
- This is because in our function InsertQ() before inserting an element, we test whether rear is equal to Q_SIZE-1. If so, we say "Queue is full and cannot insert".
- This disadvantage can be overcome using 2 methods.

a) Shift left : After deleting the element from the front, shift all remaining elements to the left.

b) Using circular representation:

- In linear queue, we increement rear by 1 and then insert an item as

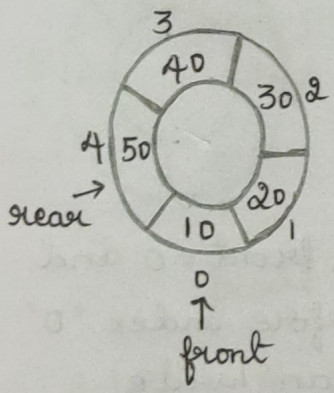
```
rear = rear + 1;
q[rear] = item;
```

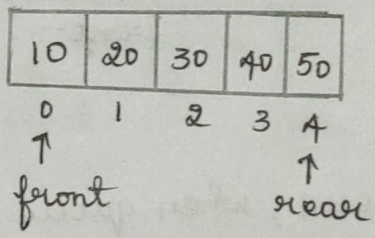
- In circular representation, we increement rear by 1 and then perform modulus operation using operator "%" as

```
rear = (rear + 1) % Q_SIZE;
```

Circular Queue:-

- In circular queue, elements of a given queue can be stored efficiently in an array so as to 'wrap around' so that end of the queue is followed by the front of queue.
- Pictorial representation of a circular queue & equivalent representation using an array is as follows:

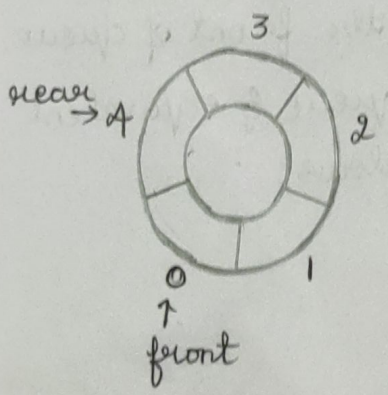


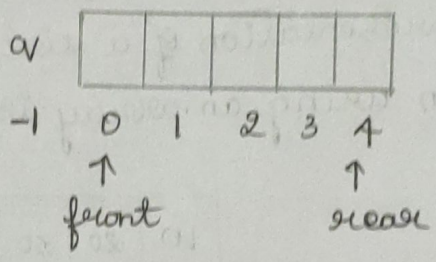


- circular representation allows the entire array to store the elements without shifting any data within the queue
- This is an efficient way of implementing queues.
- circular queue overcomes the drawback of ordinary queue
- In ordinary queue, if (rear = Q_SIZE-1) insertion is not possible, whereas in circular queue we can insert the data in circular fashion.
- Memory is used optimally.

- Before insertion, Value of rear is calculated as
  rear = (rear + 1) % Q_SIZE
- After deletion, value of front is calculated as
  front = (front + 1) % Q_SIZE
- To check overflow and underflow, global variable 'count' is maintained.
  if (count == Q_SIZE) → queue is full
  if (count == 0) → queue is empty
- Here 'count' keeps track the no of elements in the queue when no elements are there in queue, the number of elements will be '0' and hence count is initialized to '0'

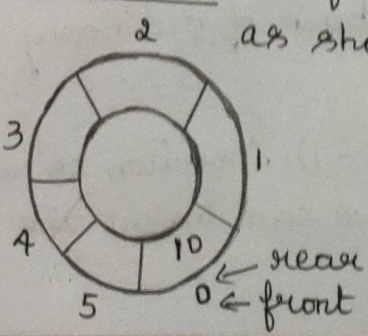
1) Empty queue :-

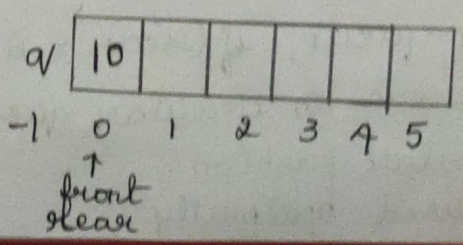




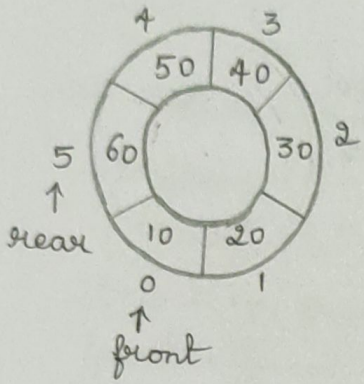
- In ordinary queue, when queue is empty, front = 0 and rear = -1. But in circular queue, just before index '0' we have '4'. So instead of rear = -1, we can write rear = 4 also, Thus empty queue is represented as
  front = 0;
  rear = Q_SIZE - 1;

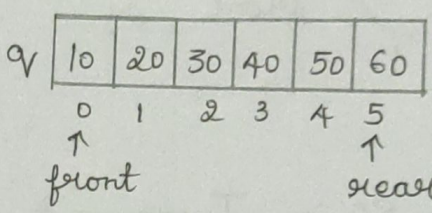
2) Insert 10 :- After incrementing rear by 1, 10 is inserted as shown below.





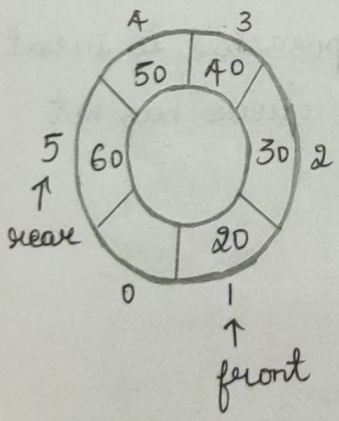
3) Insert 20, 30, 40 & 50, 60 :- Increment rear by 1 and insert 20. Again increment rear by 1 and insert 30. Again increment rear by 1 and insert 40 and so on. using the same procedure 50 & 60 will also be inserted.

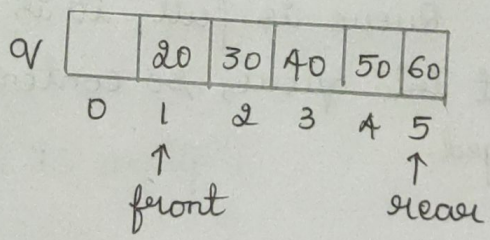




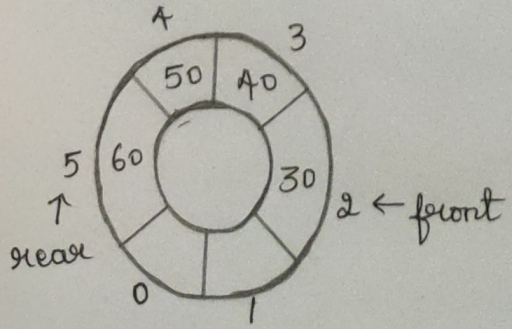
4) Inserting 80 :- Queue is full. It is not possible to insert any element into queue, so contents of queue has not been changed.

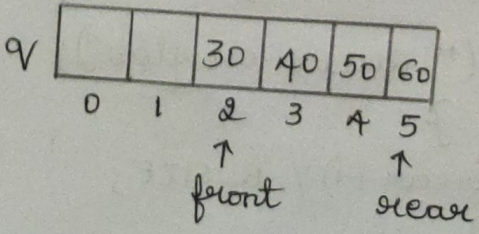
5) Delete :- An item has to be deleted always from front end so, 10 is deleted and contents of queue after deleting 10 is as shown.



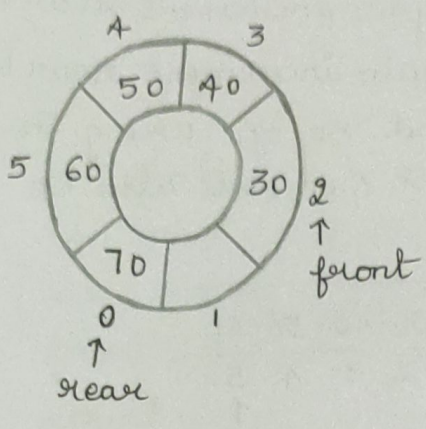


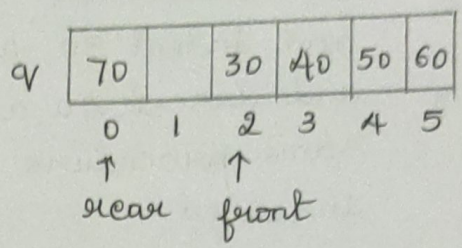
6) Delete :- An item has to be deleted always from front end. So, 20 is deleted and contents of queue after deleting 20 is as shown.



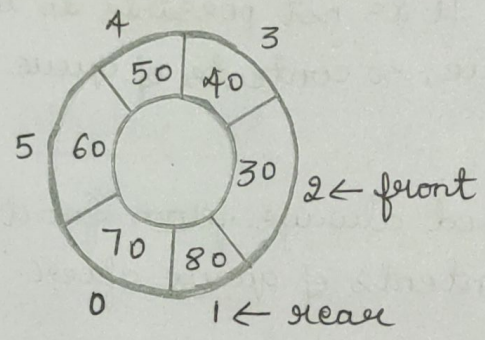


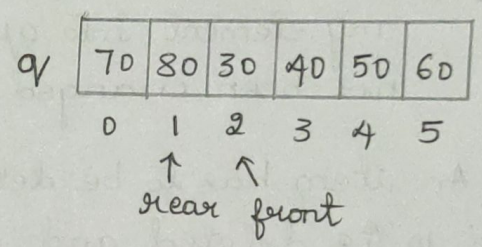
7) Inserting 70 :- Increment rear by 1 and insert 70.





8) Inserting 80 :- Increment rear by 1 and insert 80.





9) Inserting 90 :- Queue is full, it is not possible to insert any element into queue, so contents of queue has not been changed.

Operations on Circular Queue :-

1) Insert Q() :-

Function to insert an element :-

```
Void InsertQ()
{ if (count == Q_SIZE)
{ printf ("Queue overflow");
  return; }
rear = (rear+1) % Q_SIZE;
q[rear] = item;
} count++;
```

2) DeleteQ() :-

Function to delete an element :-

```
int DeleteQ()
{
  int item;
  if (count==0) return -1;
  item = q[front];
  front = (front+1) % Q_SIZE;
  count--;
  return item;
}
```

3) Display() :-

Function to display an element :-

```
Void display()
{
  int i;
  if (count==0)
{
  printf("Queue is empty");
  return; }
  printf("contents of queue are\n");
  for (i=front; i<count; i++)
  {printf("%d\n", q[front]);
   front = (front+1) % Q_SIZE;
  }
}
```

Circular Queues using Dynamic Arrays :-

- To add an element to a full queue, we must first increase the size of the queue implemented using arrays by using a dynamic memory allocation function such as malloc, calloc and realloc.

Various operation that can be performed on circular queue using dynamic arrays are :-

1) create() :-

Let us assume Q_SIZE = 1

```
int Q_SIZE = 1;
```

- Do not define Q_SIZE as a constant using #define, because using dynamic arrays, we increase Q_SIZE to accomodate more elements. So it must be a variable.
- q is a pointer and holds the address of memory location that are created using MALLOC() and CALLOC().

```
int *q;
```

- Variables used to implement circular queue are

```
int front = 0, rear = -1, count = 0;
```

∴

```
int Q_SIZE = 1;
int *q, front = 0, rear = -1, count = 0;
```

2) IsFull() :-

Same as IsFull() used in circular queue using arrays

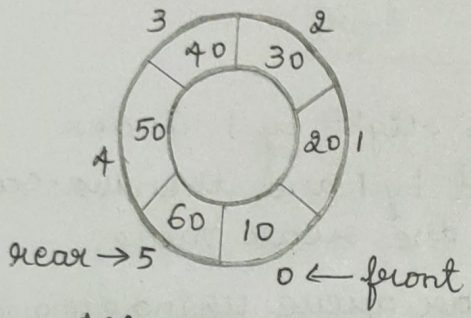
3) InsertQ() :-

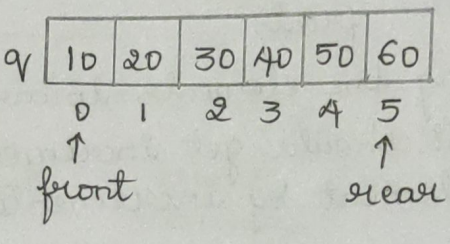
- Before inserting, we check whether sufficient space is availabel in the queue using IsFull()
- But once the queue is full, we can increase the size of array using realloc().

• However, it is not sufficient to simply increase the size using realloc(). The queue contents have to be re-adjusted.

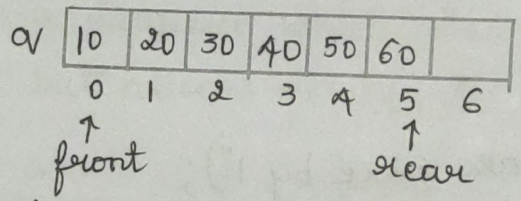
Case-1:- Front index is less than the Rear index

• consider the circular queue with 5 elements whose capacity Q_SIZE = 5





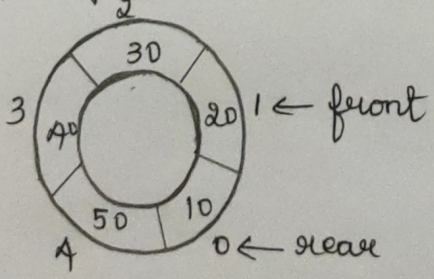
• After increasing Q_SIZE by 1 using realloc() contents of circular queue are

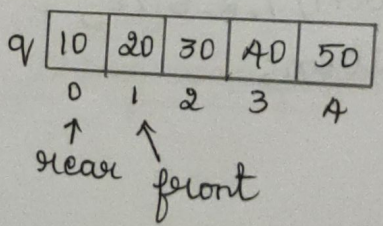


• Now an item can be inserted as usual into circular queue in usual manner by incrementing rear by 1, insert the item and then update the count by 1.

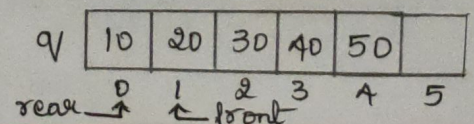
Case-2:- Front index is greater than rear index

• consider the circular queue with 5 elements whose capacity Q_SIZE = 5



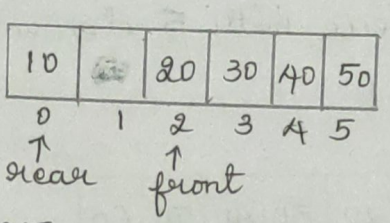


• After increasing Q_SIZE by 1 using realloc(), the contents of circular queue are



- Now to get the proper circular queue configuration, slide the elements 20, 30, 40 & 50 in the queue to the right by one position using the statement

  q[i+1] = q[i] for i = Q_SIZE − 2, where Q_SIZE = 6 after increasing the size



- After shifting the elements towards right by 1, index value of front should get incremented by 1 and then we can insert new element by incrementing the rear value.

Function to insert an item into circular queue using dynamic arrays:-

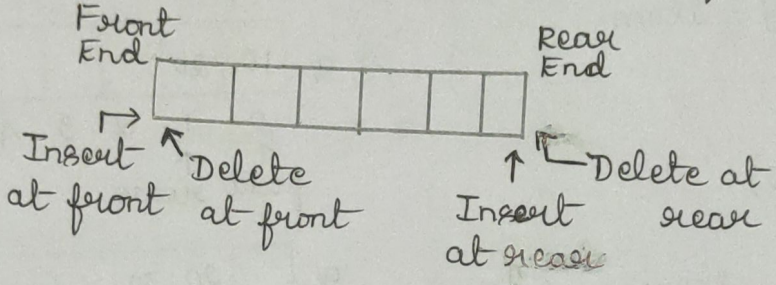
```
Void InsertQ()
{
 if (count == Q_SIZE)
 {
 printf("Q full - Increase size by 1");
  Q_SIZE++;
 q = (int *) realloc (Q_SIZE, sizeof(int));
 if (front > rear)
 { for (i = Q_SIZE - 2; i >= front; i--)
 { q[i+1] = q[i]; }
    front++;
 }
    rear = (rear+1) % Q_SIZE;
    q[rear] = item;
    count++;
}
```

4) IsEmpty()
5) DeleteQ()
6) Display()

} same as IsEmpty(), DeleteQ() and Display() used in circular queue using arrays.

Dequeues [Double Ended Queues] :-

- Dequeue is a linear list in which elements can be added or removed at either end, but not in the middle of the queue.
- Insertion is possible at both front and rear end, similarly deletion can be done at both front and rear end.



There are 2 variations of a dequeue :-

1) Input restricted dequeue :-

- It is a dequeue which allows insertions at only one end of list but allows deletion at both ends of the list.

2) Output restricted dequeue :-

- It is a dequeue which allows deletion at only one end of list but allows insertion at both ends of the list.

Operations performed on dequeue :-

1) Insert front()
2) Insert Rear()
3) Delete front()
4) Delete Rear()
5) Display()

Implementation of Dequeues using arrays :-

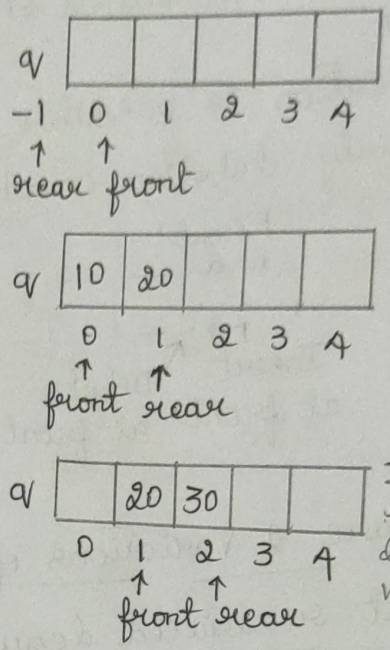
1) create() :-

```
#define Q_SIZE 5
int q[Q_SIZE];
int front = 0, rear = -1;
```

2) Insert front() :-

- Insertion of an element at front end of queue.

```
Void insertfront()
{
  if (front==0 && rear==-1)
{
  q[++rear] = item;
  return;
}
if (front!=0)
{
  q[--front] = item;
  return;
}
printf("Insertion at front is not possible");
}
```



3) Insert rear() :-

- Same as insertion of an element at rear end in linear queue or ordinary queue.

4) Delete front() :-

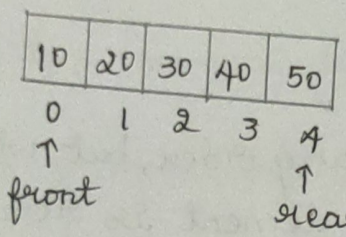
- Same as deletion of an element at front end in linear queue or ordinary queue.

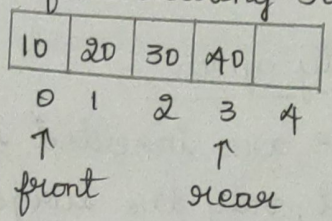
5) Delete rear() :-

- Deletion of an element at rear end of queue.

```
Void deleterear()
{
  if (front > rear)
  {
    printf("Queue Underflow");
```

```
    front = 0, rear = -1;
    return;
}
printf("Element deleted = %d\n", q[r--]);
}
```





After deleting 50 from rear end by decrementing the rear value.

6) Display():-

```
Void display()
{ int i;
 if (front > rear)
{ printf("Queue underflow");
  front = 0, rear = -1; return;}
 printf("contents of queue");
 for (i = front; i <= rear; i++)
  printf("%d\n", q[i]);
}
```

Priority Queues:-

- A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules

a) An element of higher priority is processed before any element of lower priority.

b) Two elements with the same priority are processed according to the order in which they were added to queue.

Types of priority queue :-

1) Ascending priority queue :-

- Here elements are inserted in any order, but while deleting an item from the queue, only the smallest element is removed first.

2) Descending priority queue :-

- Here also elements are inserted in any order, but while deleting an element, only the largest element is removed first.

Representation of priority queue :-

1) one way list representation of priority queue :-

- In this representation

i) Each node in the list will contain 3 items of information i.e, an information field INFO, priority number PRN and a link number LNK

ii) A node X preceeds a node Y in the list

a) when X has higher priority than Y

b) when both have same priority but X was added to the list before Y. This means that order in the one-way list corresponds to order of priority queue.

Eg :- Figure shows a schematic diagram of priority queue with 7 elements.

- Diagram does not tell us whether B was added to list before or after D. But it tells us that B was inserted before C, because B and C have same priority number and B appears before C in list.

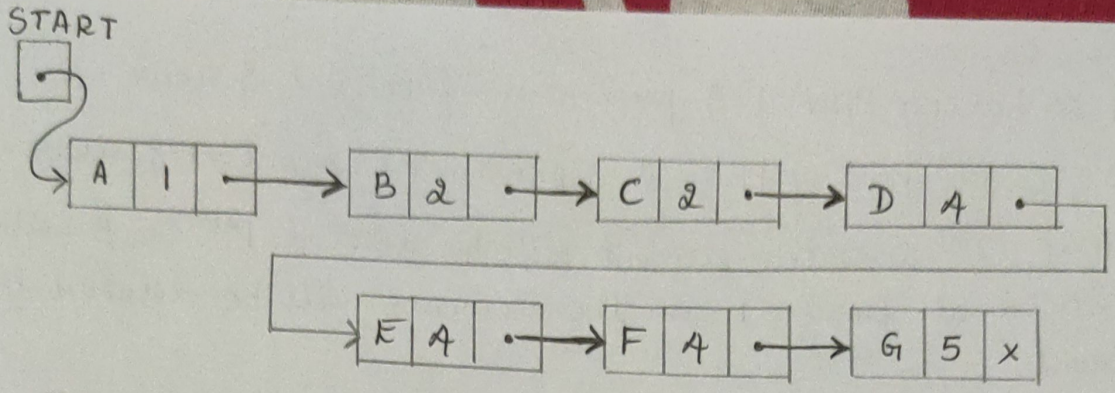


Fig :- One - way list representation of priority queue.

② Array representation of priority queue :-

- Another way to maintain priority queue in memory is to use a separate queue for each level of priority or for each priority number.
- Each such queue will appear in its own circular array and must have its own pair of pointers, FRONT & REAR.
- If each queue is allocated the same amount of space a 2D-array queue can be used instead of linear arrays.

Eg :-

INFO	PRN
A	1
B	2
C	2
D	4
E	4
F	4
G	5

PRN	FRONT	REAR
1	2	2
2	1	2
3	0	0
4	5	1
5	4	4

Priority Numbers	1	2	3	4	5	6 → Front and Rear values
1		A				
2	B	C				
3						
4	F				D	E
5				G		

Fig :- Array representation of priority queue.

- A is having PRN = 1 & present at front = 2 & rear = 2
- B & C are having PRN = 2 & present at front = 1 & rear = 2
- Here, 1st inserted element will be deleted 1st. So, B will be placed at front = 1 as the elements will be deleted from front.
- Next inserted element will be placed at rear = 2 as new elements are inserted at rear end by incrementing it.
- D, E & F are having PRN = 4 & present at front = 5 and rear = 1
- Here, 1st inserted element will be deleted 1st. So, D will be placed at front = 5 as the elements will be deleted from front.
- After inserting D, E and F will be inserted in a circular manner i.e, after 5, E will be inserted at 6 and F will be inserted at 1 by incrementing rear value where insertion occurs.
- G is having PRN = 5 & present at front = 4 & rear = 4.