DATA STRUCTURES

STACK(Last In First Out data structure)

- Stack is a data structure in which the elements are inserted and deleted from the same end called *top*.
- The element inserted last is the one to be deleted first.
- The insert operation is called PUSH and delete operation is called POP

Examples with diagrams will be discussed in the class

Implementation

1. PUSH

```
void PUSH(int *top, int item)
{
        if(*top==MAXSIZE-1)
                {cout<<“Stack full\n”;
                 return;
                }
        stack[++*top]=item;
}
```

2. POP

```
int POP(int *top)
{
        if(*top==-1)
                { cout<<“Stack underflows\n”
                  return -1;
                }
        return stack[(*top )--];
}
```

QUEUE(First In First Out data structure)

- Queue is a data structure in which the elements are inserted at one end called *rear* and delete from another end called *front.*
- The element inserted first is the one to be deleted first

Examples with diagrams will be discussed in the class

Implementation

1. **ADD**

```
void ADDQ(int *rear, int item)
{
        if(*rear==MAXSIZE-1)
                {cout<<“Queue full\n”;
                 return;
                }

        queue[++*rear]=item;
}
```

2. **DELETE**

```
int DELETEQ(int *front, int rear)
{
        if(rear==*front)
                { cout<<“Queue empty\n”
                  return -1;
                }
        return (queue[++*front]);
```

```
    }
```

CIRCULAR QUEUES

In linear queue, once we reach MAXSIZE-1, further elements can not be added eventhough there are free locations in the front.

Initially *front*=*rear*=0

Discuss examples for different situations with 6 locations:

1. Empty queue(front=rear=0)
2. Front=0, rear=3
3. Queue full(front= 0, rear=5)
4. front=4, rear=3

rear- current end of the queue
front- always points one position counter clockwise from I element in the queue

Adding element into Circular Queue

```
void ADDCQ(int front, int *rear, int item)
{
    *rear=(*rear+1)%MAXSIZE;
    if(front==*rear)
    {
        queue_full(rear);
        return;
    }
    queue[*rear]=item;
}

void queue_full(int *rear)
{
    *rear=(*rear+(MAXSIZE-1))%MAXSIZE;
    return;
}
```

Delete an element from a circular queue

```
int DELETECQ(int *front, int rear)
{
        if(*front==rear)
                cout<<"CQ empty";
        else
        {
                *front=(*front+1)%MAXSIZE;
                return queue[*front];
        }
 }
```

DOUBLE ENDED QUEUES(Dequeues)

Discuss examples

Implementation

INSERT FRONT

```
void inert_frontdq(int *rear, int *front, element item)
{
        if(*rear==-1 && *front==-1)
                        queue[++(*rear)]=item;
        else if(*front!=-1)
                queue[(*front)--]=item;
        else
                cout<<"Front insertion not possible\n");
}
```

DELETE REAR

```
element delete_reardq(int *front, int *rear)
```

```
{
        if(*front==*rear)
        {
                cout<<" queue underflows\n"
                *front=-1;
                *rear=-1;
                return-1;
        }

        return queue[(*rear)--];
}
```

Priority Queue: Elements are inserted in random order but deleted in a particular order

- Ascending priority queue
- Descending priority queue

Infix, postfix and prefix expressions

Infix: *operand1 operator operand2* - Ex: A+B
Prefix: *operator operand1 operand2* - Ex: +AB
Postfix: *operand1 operand2 operator* - Ex: AB+

Conversion from one form to another

Examples:

	Infix	Postfix	Prefix
i.	A+B-C	AB+C-	-+ABC
ii.	(A+B)*(C-D)	AB+CD-*	*+AB-CD

Exercises:

1. Convert the following infix expressions into postfix and prefix
    i. A$B*C-D+E/F/(G+H)
       Postfix
       A$B*C-D+E/F/GH+
       AB$*C-D+E/F/GH+
       AB$C*-D+E/F/GH+
       AB$C*-D+EF//GH+

AB$C*-D+EF/GH+/
AB$C*D-+EF/GH+/

AB$C*D-EF/GH+/+

Prefix
A$B*C-D+E/F/+GH
$AB*C-D+E/F/+GH
*$ABC-D+E/F/+GH
*$ABC-D+/EF/+GH
*$ABC-D+//EF+GH
-*$ABCD+//EF+GH
+-*$ABCD//EF+GH

ii. ((A+B)*C-(D-E))$(F+G)
Postfix: AB+C*DE--FG+$
Prefix: $-*+ABC-DE+FG

iii. A-B/(C*D$E)
Postfix: ABCDE$*/-
Prefix: -A/B*C$DE

iv. A/B-C+D*E-A*C

v. A+(((B-C)*(D-E)+F)/G)$(H-J)

Applications of Stacks

- Evaluation of expressions

In order to evaluate a postfix expression we use a stack

Ex: Infix 6/2-3+4*2

Postfix 6 2 /3 – 4 2 * +

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

Function to evaluate postfix expression

```
#define MAX_STACK_SIZE 100
#define MAX_EXPR_SIZE 100
int stack[MAX_STACK_SIZE];
char expr[MAX_EXPR_SIZE];

int eval(void){
        char t;
        char symbol;
        int op1, op2;
        int n=0;
        int top=-1;
        t=get_nextchar(&symbol, &n);
        while(t!=’ ‘)
        {
                if(t==’o’)
                        PUSH(&top, symbol-‘0’);
                else
                {
                        op2=POP(&top);
                        op1=POP(&top);
                        switch(t){
                                case ‘+’: PUSH(&top, op1+op2);break;
                                case ‘-‘: PUSH(&top, op1-op2);break;
                                case ‘*’: PUSH(&top, p1*op2);break;
                                case ‘/’: PUSH(&top, op1/op2);break;
                                case ‘%’: PUSH(&top, op1%op2);break;
                        }
                }

                t=get_nextchar(&symbol, &n);
        }
        return POP(&top);
}
```

The function get_nextchar()

```
char get_nextchar(char * symbol, int *n)
{
        *symbol=expr[(*n)++];
        switch(*symbol)
        {
                case ‘+‘: return ‘+’; break;
                case ‘-‘: return ‘-‘; break;
                case ‘*‘: return ‘*’; break;
                case ‘/‘: return ‘/’; break;
                case ‘%‘: return ‘%’; break;
                case ‘ ‘: return ‘ ‘; break;
                default: return ‘o’;
        }
}
```

<u>Conversion from infix to postfix</u>

Define two precedence functions *stack_prec()* and *input_prec()*.

The function *stack_prec( )*contains the precedence values of symbols on top of the stack and *input_prec()* contains the precedence values of symbols in the i/p string. The precedence values associated with these functions are shown in the following table:

symbols	**input_prec()**	**stack_prec()**
+, -	1	2
*, /	3	4
$	6	5
operands	7	8
(	9	0
)	0	-
#	-	-1

```
int input_prec(char symbol)
{
        switch(symbol)
        {
                case '+':
                case '-': return 1; break;
                case '*':
                case '/': return 3; break;
                case '$': return 6; break;
                case '(': return 9; break;
                case ')': return 0; break;
                default: return 7;
        }
}

int stack_prec(char symbol)
{
        switch(symbol)
        {
                case '+':
```

```
                case ‘-’: return 2; break;
                case ‘*’:
                case ‘/’: return 4; break;
                case ‘$’: return 5; break;
                case ‘(’: return 0; break;
                case ‘#’: return -1; break;
                default: return 8;
        }
}
```

If the operator is left associative, i/p precedence is less than the stack precedence and if an operator is right associative, i/p precedence is higher than the stack precedence.

Initial configuration

stack	**input**	**output**
#	(A+(B-C)*D)	-

Final configuration

stack	**input**	**output**
#	-	ABC-D*+

Procedure to convert an infix expression to postfix

1. Scan the next input symbol from left to right
2. As long as the precedence value of the symbol on top of the stack is greater than the precedence value of the current input symbol, pop an item from the stack and place it in the postfix expression.
   The code for this statement can be of the form:

   ```
   while(stack_prec(stack[top])>input_prec(symbol))
           postfix[j++]=pop(&top);
   end while
   ```

where the initial value of j is 0.

3. Once the condition in while loop is failed, if the precedence of the symbol on top of the stack is not equal to the precedence value of the current i/p symbol, push the current symbol on the stack. Otherwise, pop an item from the stack but do not place it in the postfix expression.
   The code for this is of the form:
   ```
   if((stack_prec(stack[top])!=input_prec(symbol))
           push(&top, symbol);
   else
           pop(&top);
   ```