Date .....

Unit-3

Stack

A stack is a linear data structure that follows LIFO principle ie Last In First Out. Stack has one end ie top from where insertion and deletion takes place. It contains one pointer which points to the top element in the stack.

Whenever an element is added on to the stack, it added on the top of stack and whenever element is deleted from the stack, it is deleted from the top of stack.

A stack is defined as a container in which insertion and deletion takes place from only one end called top of the stack.

Stack can be implemented in two ways:

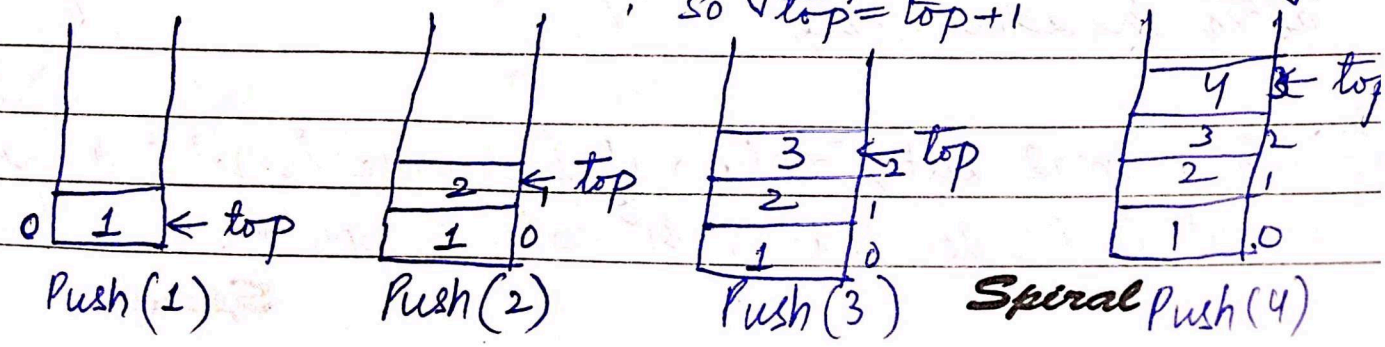
1) Static implementation
2) Dynamic implementation

Ex:- Consider a stack which is implemented statically having 5 memory blocks.

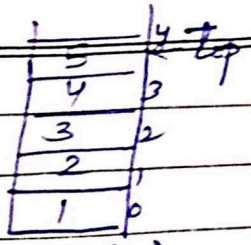
Now we have to insert 5 elements in it. We cannot insert 6th element because stack is implemented statically.

To insert: 1, 2, 3, 4, 5, Initially top = -1 (stack empty)

, so top = top+1



Date .....



Push(5)

Now stack is full and no more elements can be inserted into the stack.

Standard stack operations

1. Push() → Insertion of element at the top of stack. If the stack is full then overflow condition occurs.

2. Pop() → Deletion of element from the top of stack. If the stack is empty, then underflow condition occurs.

3. isEmpty() → It determines whether the stack is empty or not.
4. isFull() → It determines whether the stack is full or not.
5. peek() → It returns the top element of the stack
6. count() → It returns the total no. of elements in the stack.
7. change() → It changes the element at given position
8. display() → It prints all the elements available in the stack.

Static implementation of stack

```
#include <stdio.h>
#define CAPACITY 5
int stack[CAPACITY];
int top = -1;
int main()
{
int ch;
while(ch!=8)
{
printf("\n1. isEmpty()");
printf("\n2. isFull()");
printf("\n3. Insert an element");
printf("\n4. Delete an element");
printf("\n5. Peek");
printf("\n6. Display");
printf("\n7. Count");
printf("\n Enter your choice");
scanf("%d", &ch);
switch(ch)
{
case 1: isEmpty();
        break;
case 2: isfull();
        break;
case 3: push();
        break;
case 4: pop();
        break;
case 5: peek();
        break;
```

Spiral

```
case6: display();
       break;
case7: count();
       break;
default: printf("\n Wrong choice");
 }
}
return 0;
}
int isEmpty()
{
 if (top == -1)
 {
  printf("\n Stack is empty");
  return 1;
 }
 else
 {
  return 0;
 }
}
int isFull()
{
 if (top == CAPACITY-1)
 {
  printf("\n Stack is full");
  return 1;
 }
 else
 {
 return 0;
 }
}
```

Spiral

Date .....

```
void push()
{  int item; printf("\n Enter item to be inserted"); scanf("%d",
                                                    &item);
  if(!isFull())
  {
  top = top + 1;
  stack[top] = item;
  printf("\n Item inserted successfully");
    }
    else
    {
     isFull();
    }
}
void pop()
{
 if(!isEmpty())
  {
  printf("%d is deleted from stack", stack[top]);
   top = top - 1;
   printf("\n Deletion is successfull");
   }
   else
   {
    isEmpty();
   }
}
  void peek()
  {
   if(!isEmpty())
    {
    printf("%d is top element of the stack", stack[top]);
     }
```

Spiral

Date .....

```
else
{
isEmpty();
}
}
void count()
{
int count; count = 0;
for(int i = 0; i<CAPACITY; i++)
{
    count = count + 1;
}
printf("\n Total no of elements in stack are %d", count);
}
void display()
{
if(!isEmpty())
{
for(int i = 0; i< top ; i++)
{
printf("%d\t", stack[i]);
}
}
else
{
isEmpty();
}
}
```

Date .....

Dynamic Implementation of Stack

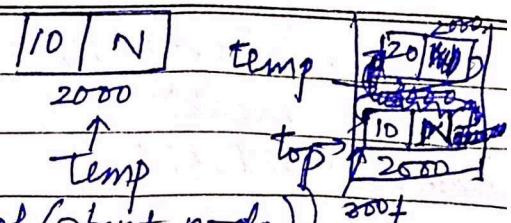
```
#include<stdio.h>
#include<stdlib.h>
  struct node
  {
   int data;
   struct node *next;
   };
  struct node *root; int isEmpty();
  void push();
  void pop();
  void peek();
 void traverse();
 void count(); struct node *top;
 int main()
 {
  int ch;
  while(ch!=7)
  {
  printf("\n1. IsEmpty());
  printf("\n2. Push");
  printf("\n3. Pop");
  printf("\n4. Peek");
  printf("\n5. Traverse");
  printf("\n6. Count");
 printf("\n Enter your choice");
  scanf("%d",&ch);
 switch(ch)
 {
```

Date .....

```
case 1: isEmpty();
        break;
case 2: isFull();
        break;
case 2: push();
        break;
case 3: pop();
        break;
case 4: peek();
        break;
case 5: traverse();
        break;
case 6: count();
        break;  default: printf("\n Wrong choice");
}
}
return 0;
}

int isEmpty()
{
  if(top==NULL)
  {
    printf("\n stack is empty");
    return 1;
  }
else
{ return 0;
}
}
```

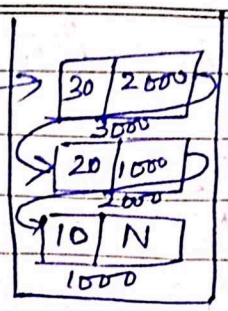
Date .....




```
void push()
{
struct node *temp;
temp = (struct node *) malloc (sizeof (struct node));
printf ("\n Enter item to be inserted");
scanf ("%d", &temp->data);
temp->next = NULL;
if (root == NULL)
{
  root = temp; top = temp;
}
else
{
temp->next = top;
top = temp;
}
}
void pop ()
{
struct node *temp;
if (!isEmpty())
{
printf ("%d is deleted from stack", top->data);
top = top->next; temp->next = NULL; free(temp);
} else {
        isEmpty();
}
}
```

Spiral

Date .....




```
void peek()
{
  if (!isEmpty())
  {
  printf("%d is top element", top->data);
  }
  else
  {
   isEmpty();
  }
}
void traverse()
  {
  if (!isEmpty())
  {
   while(top!=NULL)
   {
   printf("%d", top->data);
   top = top->next;
  } }
else
{
 isEmpty();
}
}
void count()
{
  int count;
   while(top!=NULL)
    {
     count++;
     top= top->next;
    } printf("%d Total elements are", count); }
```

Date .....

Infix, Prefix & PostFix Expressions

The way to write an arithmetic expression is called as Notation. An arithmetic expression can be expressed in three different notations :-

1. Infix Notation
2. Prefix (Polish) Notation
3. Postfix (Reverse Polish) Notation

Infix Notation

An expression in infix notation will be written as a-b+c, where operators are used in between the operands.

Prefix Notation

An expression in prefix notation will be written as +ab. Here the operator comes before the operands. Prefix Notation is also known as Polish Notation.

Postfix Notation

An expression in postfix notation will be written as ab+. Here the operator comes after the operands. This notation style is also called as Reversed Polish Notation.

Date .....

For converting any infix expression to postfix or prefix notation we should know the operator precedence

Ex:- Parenthesis operator has the highest precedence.

S.No	Operator	Precedence	Associativity
1	()	Highest	Left to right
2	^	Second Highest	Right to left
3	* / %	Third Highest	Left to Right
4	+ -	Fourth Highest	Left to Right

Associativity Example :-

1 + 2 * 3 + 30 / 5

1 + 6 + 6

7 + 6

= 13

Convert the following infix expression to prefix :

Exp : a + b * c

= a + (* b c)

= + a * b c

Exp : a * b + c

= (* a b) + c

= + * a b c

Infix expression to postfix :

Exp : a + b * c

= a + b c *

= a b c * +

Spiral

Date .....

Exp: a * b + c
= (a b *) + c
= a b * c +

Exp: a + b / c
= a + b c /
= a b c / +

Infix to Postfix Expression using Stack

Expression:- K + L - M * N + (O ^ P) * W / U / V * T + Q

First we will scan the entire expression from left to right.
If the precedence of incoming operator is higher than precedence of existing operator then we will push that operator on the top of stack.
If the precedence of incoming operator is less than precedence of existing operator then we will pop the existing operator from the stack.

Input	Stack	Postfix Expression
K	-	K
+	+	K
L	+	K L
-	-	K L +
M	-	K L + M
*	- *	K L + M
N	- *	K L + M N
+	-	K L + M N * -
(	+ (	K L + M N * -

Spiral

Date .....

O	+(	KL+MN*-O
^	+(^	KL+MN*-O
P	+(^	KL+MN*-OP
)	+	KL+MN*-OP^
*	+*	KL+MN*-OP^
W	+*	KL+MN*-OP^W
/	+*/	KL+MN*-OP^W*
U	+/	KL+MN*-OP^W*U
/	+/	KL+MN*-OP^W*U/
V	+/	KL+MN*-OP^W*U/V
*	+*	KL+MN*-OP^W*U/V/
T	+*	KL+MN*-OP^W*U/V/T
+	+	KL+MN*-OP^W*U/V/T*+
Q	+	KL+MN*-OP^W*U/V/T*+Q+

Resultant postfix Expression will be:
KL+MN*-OP^W*U/V/T*+Q+

Expression :- A+B*C+D

I/P	Stack	Expression
	-	A
A	+	A
+	+	AB
B	+*	AB
*	+*	ABC
C	+	ABC*+
+	+	ABC*+D+
D		

Date .....

Expression :- ((A+B)-C*(D/E))+F

I/p	Stack	Expression
(	(	-
(	((	-
A	((	A
+	((+	A
B	((+	AB
)	(	AB+
-	(-	AB+
C	(-	AB+C
*	(-*	AB+C
(	(-*(	AB+C
D	(-*(	AB+CD
/	(-*(/	AB+CD
E	(-*(/	AB+CDE
)	(-*	AB+CDE/
)		AB+CDE/*-
+	+	AB+CDE/*-
F	+	AB+CDE/*-F+

Date .....

Infix to Prefix Expression using Stack

Expression :- K+L-M*N+(O^P)*W/U/V*T+Q

First of all reverse the entire expression

Exp.:2

Q+T*V/U/W*)P^O(+N*M-L+K

I/P	stack	Expression
Q	-	Q
+	+	Q
T	+	QT
*	+*	QT
V	+*	QTV
/	+*/	QTV
U	+*/	QTVU
/	+*//	QTVU
W	+*//	QTVUW
*	+*//*	QTVUW
)	+*//*)	QTVUW
P	+*//*)	QTVUWP
^	+*//*)^	QTVUWP
O	+*//*)^	QTVUWPO
(	+*//*	QTVUWPO^
+	++	QTVUWPO^*//*
N	++	QTVUWPO^*//*N
*	++*	QTVUWPO^*//*N
M	++*	QTVUWPO^*//*NM
-	++-	QTVUWPO^*//*NM*
L	++-	QTVUWPO^*//*NM*L

Spiral

Date .....

+	++-+	QTVUWPO^*//*NM*L
K	++-+	QTVUWPO^*//*NM*LK

~~Final~~ QTVUWPO^*//*NM*LK+-++

Now again reverse the expression to get the final prefix expression :

++-+KL*MN*//*^OPWUVTQ

Exp:- A*B+C/D

Reverse:- D/C+B*A

I/P	Stack	Expression
D	-	D
/	/	D
C	+	DC
+	+	DC/
B	+	DC/B
*	+*	DC/B
A	+*	DC/BA

DC/BA*+

Prefix Expression will be : +*AB/CD

Date .....

Exp:- (A-B/C)*(A/K-L)

Reverse:-)L-K/A(*)C/B-A(

I/P	Stack	Expression
)	)	-
L	)	L
-	)-	LK
K	)-	LK
/	)-/	LKA
A	)-/	LKA/-
(		LKA/-
*	*	LKA/-
)	*)	LKA/-C
C	*)	LKA/-C
/	*)/	LKA/-CB
B	*)/	LKA/-CB/
-	*)-	LKA/-CB/A
A	*)-	LKA/-CB/A-*

Final prefix expression will be: *-A/BC-/AKL

Date .....

Expression :- ((a+b)+c)-(d+(e*f))

))f*e(+d(-)c+)b/a((

I/p	Stack	Exp
)	)	-
)	))	-
)	))	f
f		
*	))*	f
e	))*	fe
(	)	fe*
+	)+	fe*
d	)+	fe*d
(		fe*d+
-	-	fe*d+
)	-)	fe*d+
c	-)	fe*d+c
+	-)+	fe*d+c
)	-)+)	fe*d+c
b	-)+)	fe*d+cb
/	-)+)/	fe*d+cb
a	-)+)/	fe*d+cba
(	-)+	fe*d+cba/
(	-	fe*d+cba/+-

Prefix expression :- -+/abc+d*ef

Date .....

Evaluation of Prefix Expression

Expression :- a+b*c-d/e^f

Prefix :- a+b*c-d/(^ef)

= a+(*bc)-d/(^ef)

a+(*bc)-(/d^ef)

(+a*bc)-(/d^ef)

-+a*bc/d^ef

Let a=2, b=3, c=4, d=16, e=2, f=3

-+2*34/16^23

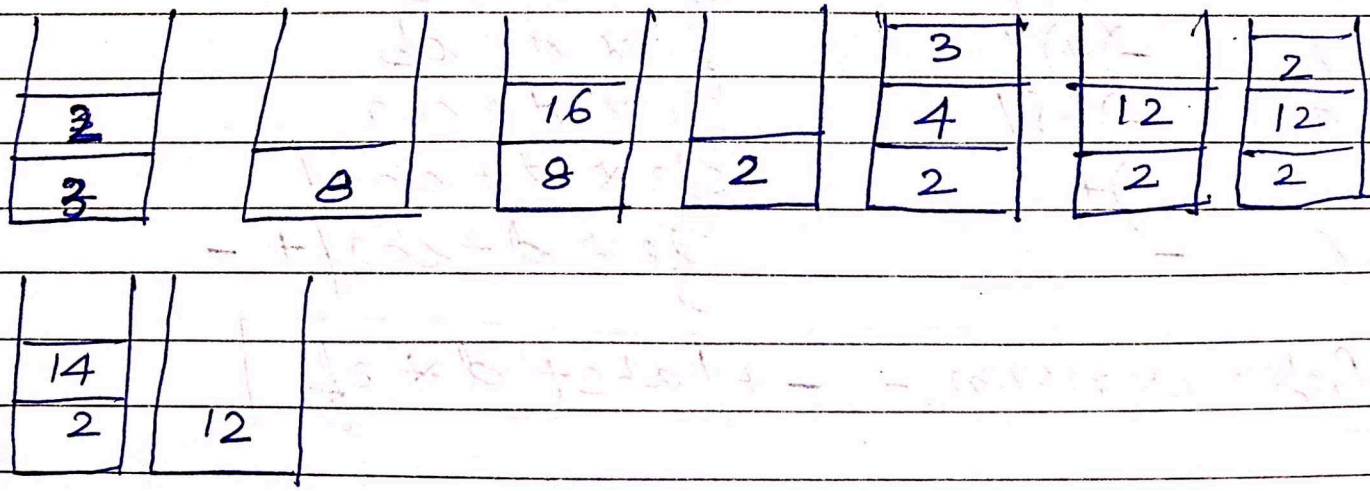
= -+2*34/16 8

= -+2 12 2

= -14 2

= 12

Evaluation Using stack



Date .....

Evaluation of Postfix Expression

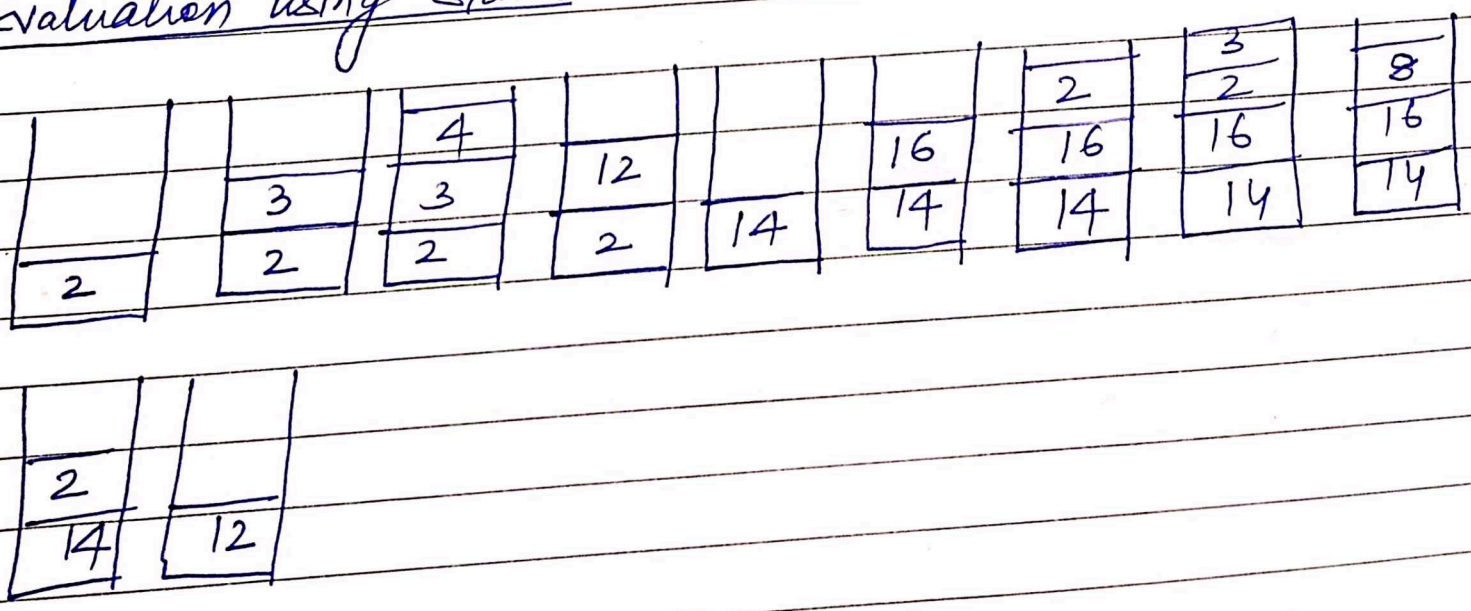
Expression :- a+b*c-d/e^f

Postfix :- a+b*c-d/(ef^)
= a+(bc*)-(def^/)
= (abc*+)-(def^/)
= abc*+def^/-

Let a=2, b=3, c=4, d=16, e=2, f=3

= 2 3 4 * + 16 2 3 ^ / -
= 2 12 + 16 2 3 ^ / -
= 14 16 2 3 ^ / -
= 14 16 8 / -
= 14 2 -
= 12

Evaluation using Stack



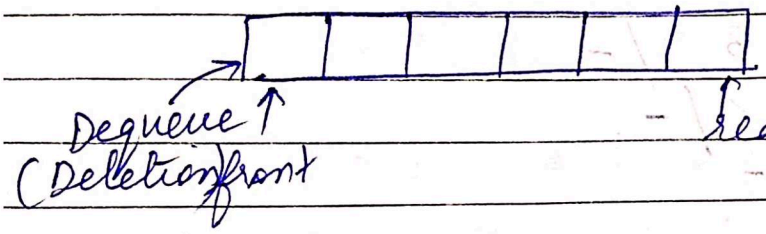
Spiral

Date .....

Queues

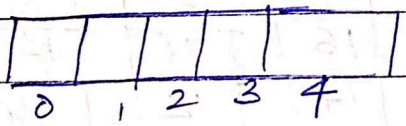
A queue is a data structure which enables insertion of elements to be performed from one end called REAR and deletion operations to be performed from another end called FRONT.
Queue is referred to as FIFO (First In First Out). The element inserted first in queue will be the first to be deleted from queue.

For ex: People waiting for bus in a queue.



Queue can be implemented using:
1. Static implementation
2. Dynamic implementation

```
int queue[5];
```



Operations on Queue

1. Insertion (Enqueue)
2. Deletion (Dequeue)
3. Display elements

Insertion of elements will be performed using rear pointer and deletion of elements will be performed using front pointer.

Date .....

Initially front and rear will be pointing to first location of array.
If rear == size. It means queue is full.
If front = rear. It means queue is empty.

~~Array implementation~~

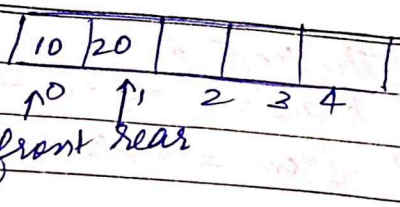
Queue implementation using Array

```
# include <stdio.h>
# define SIZE 5
int queue[SIZE];
int front = -1;
int rear = -1;
void Enqueue(); void Dequeue();
void Display();
int main()
{
int ch;
while(ch != 4)
{
printf("\n Insertion");
printf("\n Deletion");
printf("\n Display");
printf("\n Enter your choice");
scanf("%d", &ch);
switch(ch)
{
case 1: Enqueue();
        break;
Case 2: Dequeue();
        break;
Case 3: Display();
        break;
default: printf("\n Wrong choice");
}
} return 0; }
```

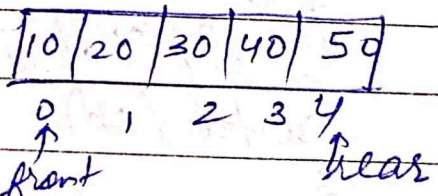
Spiral

Date .....

```
void Enqueue()
{
    int item; scanf("%d", &item);
    if (rear == SIZE-1)
    {
    printf("\n Queue is full");
    }
    else if (front == -1)
    {
      front = 0;
      rear = rear + 1;
    queue[rear] = item;
      printf("\n Item inserted successfully");
    }
    else
    {
    rear = rear + 1;
    queue[rear] = item;
    printf("\n Item inserted successfully");
    }
}
```




```
void Dequeue()
{
if(front == -1 || front > rear)
  {
    printf("\n Queue is empty");
  }
else
{
printf("\n %d is deleted from queue", queue[front]);
for(int i = 0; i < rear; i++)
```



Date .....

```
{
queue[i] = queue[i+1];
}
rear = rear-1;
}
}

void Display()
{
if (front == -1 || front > rear)
{
printf("\n Queue is empty");
}
else
{
for(int i = front ; i <= rear ; i++)
{
printf("\n%d\t", queue[i]);
}
}
```

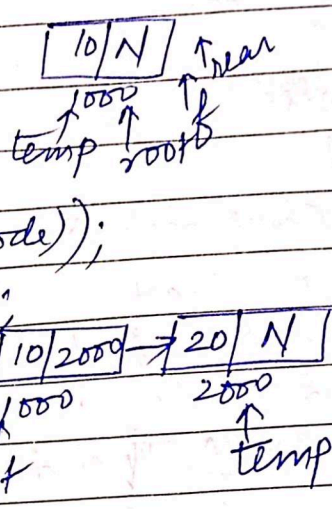
Date .....

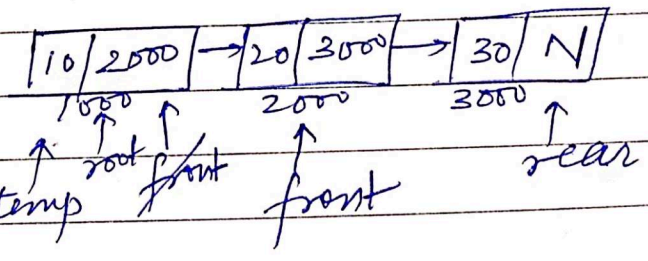
Queue implementation using Linked List

```
# include <stdio.h>
# include <stdlib.h>
  void enqueue();
  void dequeue();
  void traverse();
  struct node
{
  int data;
  struct node * next;
  };
struct node * root, * front, * rear;
  int main()
{
  int ch;
  while (ch != 4)
{
  printf("\n 1. Enqueue");
  printf("\n 2. Dequeue");
  printf("\n 3. Traverse");
  printf("\n Enter your choice");
  scanf("%d", &ch);
  switch(ch)
{
  case 1: enqueue();
          break;
  case 2: dequeue();
          break;
  case 3: traverse();
          break;
```

Date .....

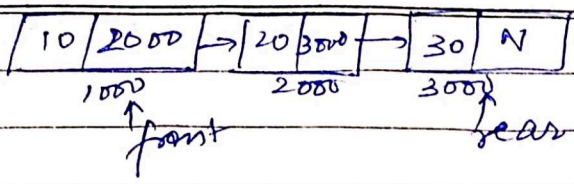
```
case 4: default: printf("\n Wrong choice");
    }
  }
  return 0;
}
void enqueue()
{
  struct node *temp;
  temp = (struct node *) malloc (sizeof(struct node));
  printf("\n Enter item to be inserted");
  scanf("%d", &temp->data);
  temp->next = NULL;
    if (root == NULL)
    {
      root = temp; front = temp; rear = temp;
    }
    else
    { rear->next = temp;
    rear = rear->next;
    }
  printf("\n Item inserted successfully");
}
void dequeue()
{
  struct node *temp;
  if (front == NULL)
  {
    printf("\n Queue is empty");
  }
  else  temp = front;
  { printf("%d is deleted from queue", front->data);
    front = front->next;
    free(temp);
  } }
```





Spiral

Date .....




```
void traverse()
{
if(front == NULL)
{
printf("\n Queue is empty");
}
else
{
struct node *p;
p = front;
while(p != NULL)
{
printf("%d->", p->data);
p = p->next;
}
}
}
```