Unit : 2

Stack :

1) Non-primitive linear Data Structure.
   (Element access in linear structure).

2) LIFO (Last in First out)

3) Operations: push() → for insertion
   pop() → for deletion
   peek() → To display topmost element of the stack.

Implementing Stack :-

→ Static Method : through Arrays → Memory allocated during compile ~~run~~ Time

→ Dynamic Method : through pointers or linked list → Memory allocated during run time.

Through Arrays :-

```
int s[10];
int tos = -1;   // Variable representing top of stack

void push()
```

```
{ int d;
  if ( tos = = 10-1)
  { printf ("Overflow");
    exit (0);
  }

else
{ scanf ("%d", &d);
   tos = tos +1;
   s[tos] = d;   <- } Can write  s[++tos]=d;
}
}
```

```
void pop()
{
  if (tos = = -1)
  { printf( "Underflow");
    exit (0);
  }
  else
  { printf("%d", s[tos]);
     tos = tos-1;
  }
}
```

```
void peek () {
printf (" %d", s[tos]);
}
```

```
void print ()
{
  int i;
 for( i= tos;  i>=0 ;  i--)
 { printf ("%d", s[i]);
  }
}
```

Implement Stack using pointer / linked list

```
Struct node {
 Int data ;
Struct node  * next ;
};
struct node  * tos = NULL;
Struct node   * temp = NULL;
int main () {}

void push ()
{ int d;
  Struct node  * ptr = NULL;
 ptr =  (struct node *) malloc (sizeof (struct node));

 if (ptr = =NULL)
{ printf ("Memory Not created"); exit (0);
 }

else {
```

```
scanf ("%d", &d);
  ptr -> data = d;
  ptr -> next = tos;
  tos = ptr;
  }
}


void pop()
{ if ( tos == NULL)
  { printf ("Underflow");
    exit (0);
  }

  else
  { temp = tos;
     printf("%d", tos -> data);
     tos = tos -> next;
     free (temp);
     temp = NULL;
  }
}


void peek()
{
  printf ("%d", tos -> data);
}
```

Recursion: function calls itself.

Direct	Indirect
A() { A(); }	A() ⇄ B() A() { B(); } B() { A(); }

Tail

```
A()
{
 ___
 ___
 ___
 A(); // func-call at last
}
```

Non-Tail
(func^n call can be done anywhere throughout the func^n).

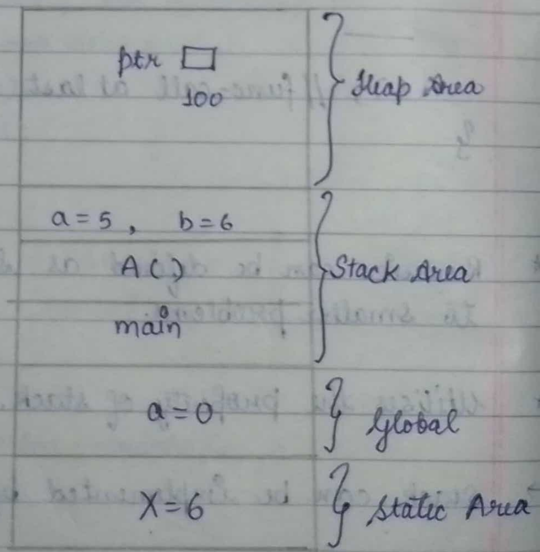
★ Recursion can be defined as solving bigger problems to smaller problems.

★ Utilises the property of stack.

→ Stack can be implemented by Recursion.

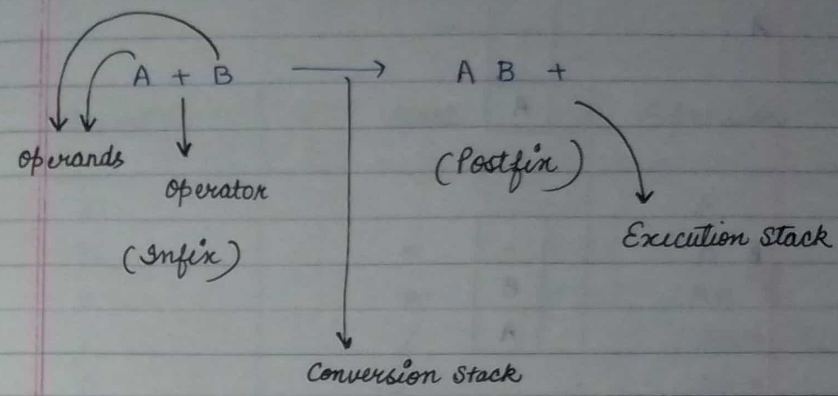
```
void A(); int a;
int main()
{ int *ptr;
  int a=5;
  auto int b=6;
  ptr= (int *)malloc(sizeof(int))
  A();
}

A()
{ static x = 7;
  x = x-1;
  A();
}
```



28.11.23

Conversion of Infix to Postfix :-



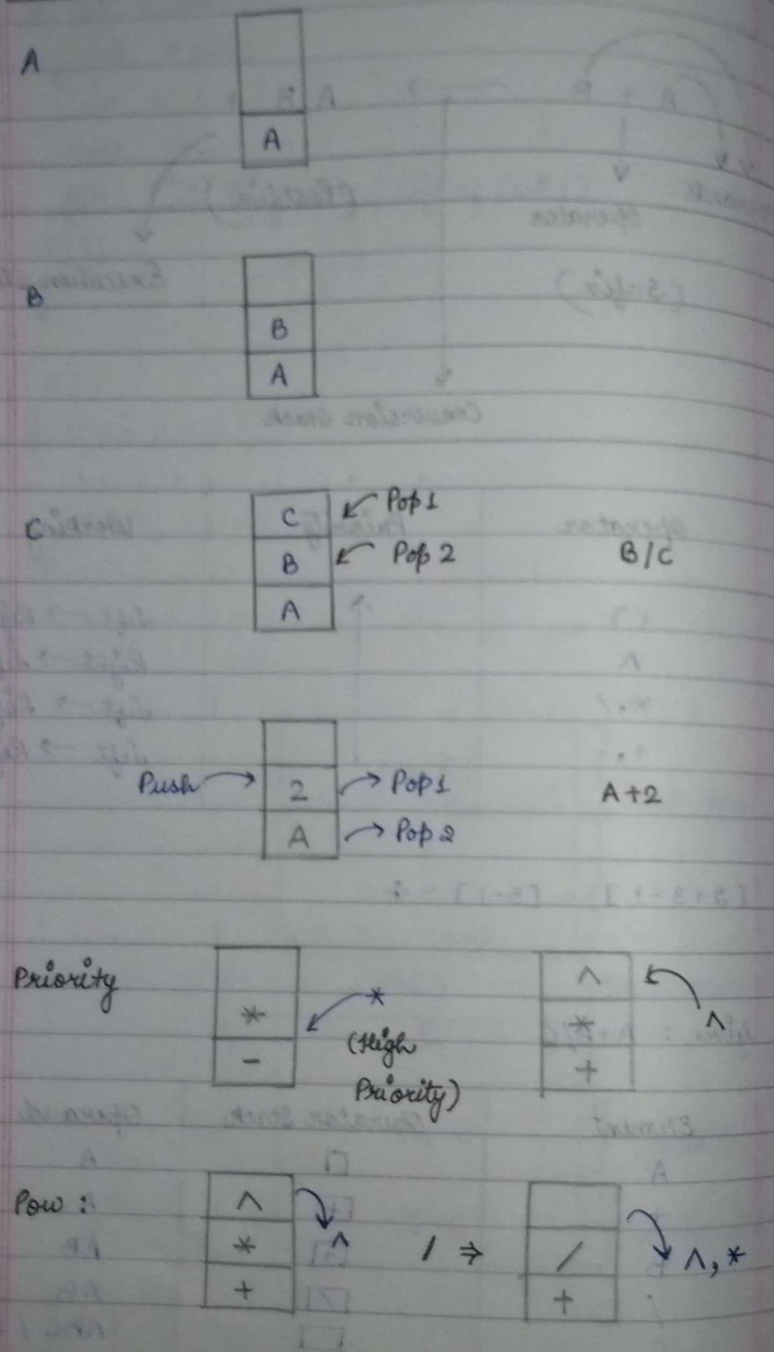
Operator	Priority	Working
( )	↑	Left → Right
^		Right → Left
*, /		Left → Right
+, -		Left → Right

[2+3-1] = [5-1] = 4

Given : A+B/C

Element	Operator Stack	Operand
A	$\boxed{\ }$	A
+	$\boxed{+}$	A
B	$\boxed{+}$	AB
/	$\boxed{/}$	AB
C	$\boxed{\ }$	$\boxed{ABC/+}$

Operand Stack



Example :-

A * B / C ^ D - E * F / H

Element	Stack	Expression
A		A
*	*	A
B	*	AB
/	/	AB* (L→R)
C	/	AB*C
^	^ /	AB*C
D	^ /	AB*CD
-	-	AB*CD^/
E	-	AB*CD^/E
*	* -	AB*CD^/E
F	* -	AB*CD^/EF
/	/ -	AB*CD^/EF*
H	/ -	AB*CD^/EF*H/-

(A+(B-C))

10	)
8	)
6	−
6	(
3 4	+
1	(

2	5	7	8	11
A	B	C	−	+

12.12.23

Queue : * Linear Data Structure
* FIFO
* Operations : Enqueue (Insertion)
Dequeue (Deletion)

front → (Deletion) and Rear → (Insertion)

Queue using array :

```
#include <stdio.h>
# define MAX_SIZE 10

int queue [MAX_SIZE];
int front = -1, rear = -1;

int isFull() {
 if (rear== MAX-SIZE -1)
  return 1;
else
  return 0;
}

int isEmpty() {
if ( front == -1 || front > rear)
  return 1;
 else
   return 0;
}

void enqueue (int value) {
 if (isFull()) {
 printf ("Queue is Full, Cannot enqueue.\n");
 }
  else {
 if ( front == -1)
 { front = 0;
  }
  rear ++;
 queue [rear] = value;
 printf (" %d enqueued \n", value);
 }
}

void dequeue () {
if (isEmpty()) {
printf("Queue is empty, Cannot dequeue.\n");
}
 else {
```

```
printf ("Element %d dequeued \n", queue[front]);
 front ++;
 }
}

int main()
{ enqueue(5);
   enqueue (10);
   enqueue (15);
   dequeue ();
   dequeue ();
   enqueue (20);
   enqueue (25);
 return 0;
 }
```

(Assignment) Implement Queue using Linked List.

```
#include <stdio.h>
# include <stdlib.h>

struct Node {
 int data;
 struct Node * next;
};

struct Node * front = NULL;
struct Node * rear = NULL;

int isEmpty()
{ return (front == NULL);
 }

void enqueue (int value) {
struct Node * newNode = (struct Node *) malloc (sizeof
(struct Node));
newNode -> data = value;
newNode -> next = NULL;

if( isEmpty()) {
  front = newNode;
  rear = newNode;
 } else {
   rear -> next = newNode;
    rear = newNode;
 }

printf ("%d enqueued \n", value);
}

void dequeue () {
 if ( isEmpty()) {
 Printf(" Queue is Empty, can't dequeue.\n");
 } else {
   struct Node * temp = front;
   printf (" Element %d dequeued \n", front -> data);
   front = front -> next;
   free (temp);
  }
}
```

```c
void display () {
 struct Node * current = front;
 if ( isEmpty())
 {printf ("Queue is Empty \n");
  return;
 }
 printf ("Queue Elements : \n");
 while ( current != NULL) {
 printf (" %d", current -> data );
  current = current -> next;
  }
  printf("\n");
}

int main()
{
  enqueue (5);
  enqueue (10);
  enqueue (15);
  display();
   dequeue ();
   display ();
 return 0;
}
```

(Assignment) Write down the applications of queue :

1) **Operating Systems** : Queues are used in scheduling algorithms (like CPU scheduling) and managing system resources (like I/O requests).

2) **Printers and Spooling** : Print jobs are placed in a queue and processed in the order they are received. Spooling systems use queues to manage data being sent to a device.

3) **Breadth First Search** : In Graph Theory, BFS uses a queue data structure to traverse or search a graph level by level.

4) **Call Center Systems** : Queues are used to manage incoming calls, where calls are placed in a queue and answered based on their arrival.

5) **Buffer Management** : Queues help manage data transmission and reception, preventing data loss or overflow in communication systems.

6) **Traffic Management** : Traffic Signals at Intersections often utilize queues to control the flow of vehicles, giving each direction its turn.

7) **Task Scheduling** : Queues can be employed to schedule tasks in various systems, ensuring fairness and order in task execution.

8) **Resource Sharing** : In multi threaded or multi process environment queues can help coordinate access to shared resources.

9) **Simulations:** Queues are used in simulations to model real-world scenarios like Customer service systems, traffic flow, etc.

10) **Asynchronous Data Transfer:** Queues facilitate asynchronous communication between components in software allowing decoupling of producers and consumers.

Circular Queue:

→ **Implementation through Array**

```
#include <stdio.h>
# define MAX_SIZE 5

int queue [MAX_SIZE];
int front = -1, rear = -1;

int isFull () {
if ((front ==0 && rear == MAX_SIZE-1) ||
  (front == rear +1 ))
  return 1;
else
   return 0;
}
```

```
int isEmpty () {
if ( front = = -1)
 return 1;
else
  return 0;
}

void enqueue( int value) {
if ( isFull ()) {
printf(" Queue is Full. Cannot Enqueue");
} else {
 if ( front = = -1)
{ front =0;
 }
rear = (rear +1) % MAX_SIZE ;
queue [rear]=value ;
printf ("%d enqueued \n", value);
}
}

void dequeue () {
if ( isEmpty()) {
printf (" Queue is Empty, cannot dequeue \n");
} else {
 printf( "Element %d dequed \n", queue [front]);
 if (front == rear) {
  front = -1;
  rear = -1;
 }
 else {
```

```c
front = (front +1) % MAX_SIZE;
  }
 }
}

void display (){
if (isEmpty ()) {
printf(" Queue is empty, \n");
 return;
}

printf ("Queue elements: ");
int i= front;
do {
    printf (" %d, queue [i] );
    i= (i+1) % MAX_SIZE;
} while (i! = (rear +1) % MAX_SIZE;
 printf ( "\n");
}

int main() {
enqueue (5);
enqueue (10);
enqueue (15);
enqueue (20);
enqueue (25);
enqueue (30);
display ();
dequeue ();
dequeue ();
```



```c
display ();
return 0;
}
```

Assignment) Implement Circular Queue using linked list.

```c
# include <stdio.h>
# include <stdlib.h>

Struct Node {
 int data;
 Struct Node *next;
};
Struct Node *front = NULL , *rear = NULL;
int is Empty () {
 return (front == NULL);
}

Struct Node * createNode (int value) {
 Struct Node * new Node = (Struct Node *) malloc (sizeof
(Struct Node));
if (newNode == NULL) {
  printf (" Memory Allocation failed \n");
  exit (1);
 }
 new Node -> data = value;
 new Node -> next = NULL;
 return newNode;
}
```

PAGE NO.:
DATE: / /20

```
void enqueue( int value) {
Struct Node * newNode = createNode (value);
if ( isEmpty()) {
   front = newNode;
} else {
    rear -> next = newNode;
 }
 rear = newNode;
 rear -> next = newNode;
 }
 rear = newNode;
 rear -> next = front;

printf(" %d enqueued \n", value);
 }

 void dequeue () {
  if (isEmpty()) {
 printf(" Circular Queue is Empty, Cannot dequeue \n");
  return;
 }

 Struct Node * temp = front;
 printf (" %d dequeued \n", front -> data);

if ( front == rear) {
  front = NULL;
  rear = NULL;
} else {
  front = front -> next;
```

PAGE NO.:
DATE: / /20

```
 rear -> next = front;
 }
 free (temp);
}

void display () {
 if (isEmpty())
 { printf (" Empty \n"); return;
 }
 Struct Node * temp = front;
 printf (" Circular Queue Elements : \n");
do {
    printf (" %d ", temp -> data);
     temp = temp -> next;
 } while ( temp != front);

 printf ("\n");
}

int main()
{ enqueue (5);
  enqueue (10);
  enqueue (15);
  enqueue (20);
  display ();
  dequeue ();
  dequeue ();
  display();
  return 0; }
```

12|12|2023

Unit : 3

Graph : ★ Set of Vertices and edges
★ Cyclic in Nature.

★ Can be implemented by Matrix (2D Array) or linked list

Tree : ★ Set of nodes and edges.
★ Non-Cyclic in Nature.

NOTE : Every tree is a Graph, but not every Graph is a tree.

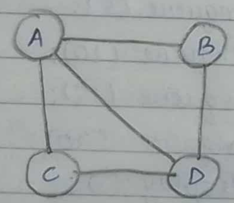
13|12|23

Representation of graph in memory :-

→ Matrix (2D Array) [Used for Dense Graph]
→ Linked List [Used for Sparse Graph]

X	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	1
D	1	1	1	0

Matrix Representation



Graph

A → B → C → D
B → A → D
C → A → D
D → A → B → C

Linked List Representation

Traversal of Graph / tree :

1) DFS : Stack data structure (Depth first Search)

2) BFS : Queue data structure (Breadth first Search)